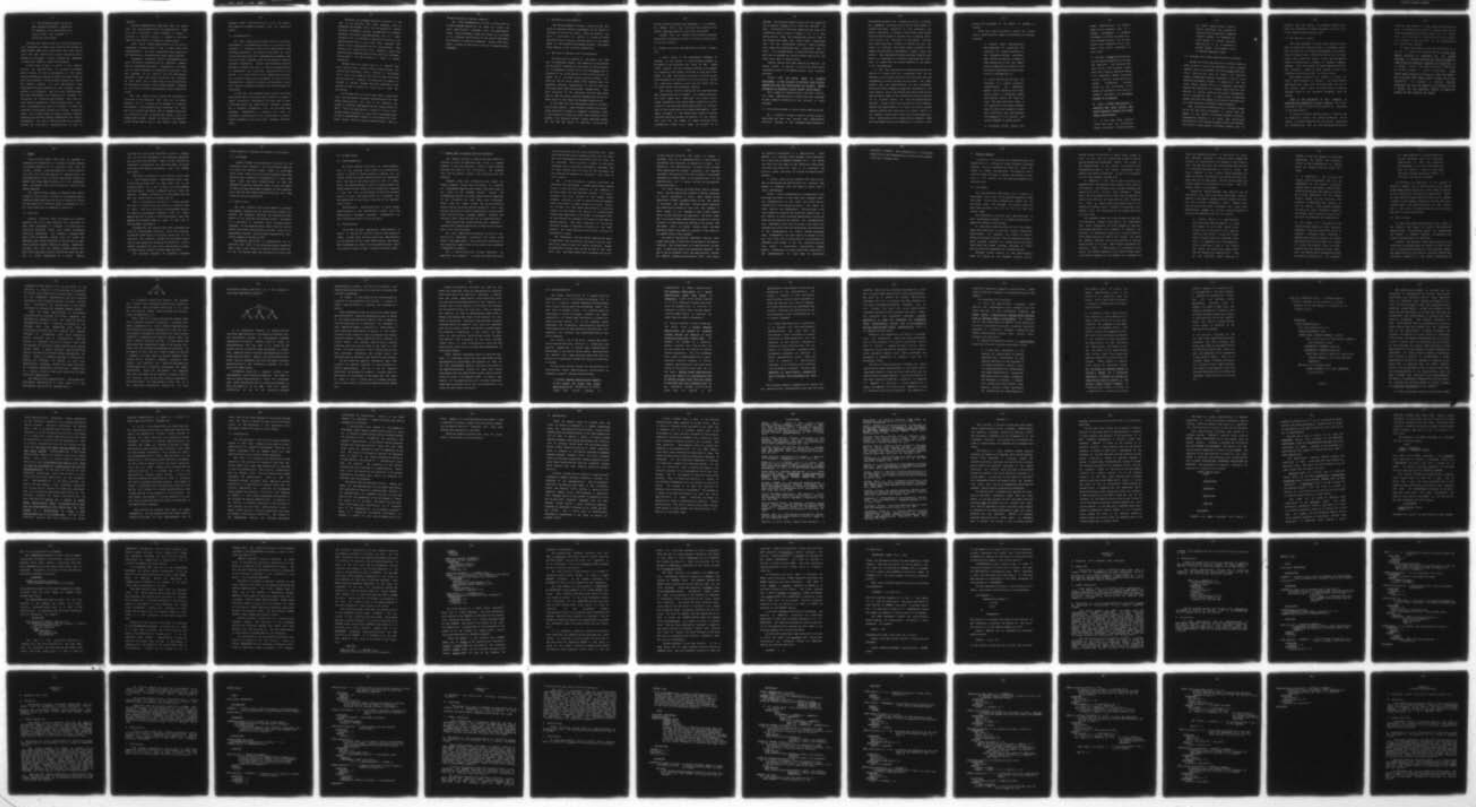
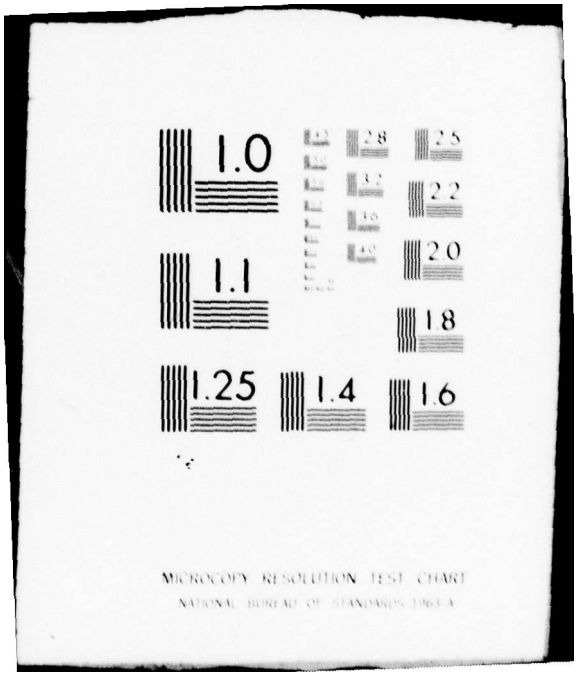
AD-A068 970 DELAWARE UNIV NEWARK DEPT OF COMPUTER AND INFORMATI--ETC F/G 9/2
TOWARD A LIBRARY OF FORMAL DESIGNS OF SOFTWARE.(U)
1979 R M WEISCHEDEL, L SALSBURG AFOSR-78-3539
UNCLASSIFIED AFOSR-TR-79-0528 NL

1 OF 2
AD A068970




1.0
1.1
1.25
1.4
1.6
2.8
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

AFOSR-TR- 79-0528

LEVEL II

AD A068970

DDC
RECEIVED
MAY 24 1979
C

Department of
COMPUTER AND INFORMATION SCIENCES

DDC FILE COPY

UNIVERSITY OF DELAWARE
Newark, Delaware 19711

Approved for public release;
distribution unlimited.

79 05 18 097

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFOSR-TR-79-0528	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): TOWARD A LIBRARY OF FORMAL DESIGNS OF SOFTWARE	5. TYPE OF REPORT & PERIOD COVERED: Final rept.
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Ralph M. Weischedel and Linda Salsburg	8. CONTRACT OR GRANT NUMBER(s): AFOSR-78-3539
9. PERFORMING ORGANIZATION NAME AND ADDRESS: University of Delaware, Department of Computer and Information Sciences, Newark, Delaware 19711	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office of Scientific Research/NM, Bolling AFB, Washington, DC 20332	12. REPORT DATE: 1979
	13. NUMBER OF PAGES: 103
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): 2304 / A2	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Many of the problems of large software systems have been attributed to the design phase of software development, problems such as high maintenance costs, the predominance of software error types traceable to design (rather than to coding), and the high cost of diagnosing and correcting design errors. The most promising approach to these problems is the formal specification of module interfaces, during the design phase, based on the information-hiding principle.

The advantages of formal specifications are as follows: (1) Their precision,

DD FORM 1 JAN 73 1473

UNCLASSIFIED 411182

20. Abstract continued

lack of ambiguity, and attention to detail should cut down on design errors. (2) They provide informal verification of a hierarchically designed system while it is being designed. (3) Special design validation teams could rigorously verify a design before it is implemented, perhaps with the aid of automated tools for some of the verification. (4) Formal specification enables rigorous specification of the requirements that an embedded computer system must conform to. (5) They combine with the information-hiding principle to enable design of systems that are much easier to modify and maintain.

Yet, the formal specification of modules itself has a serious drawback: the upfront effort in creating them is considerable. Consequently, this research has investigated the feasibility of a library of formal specifications so that designers could build on the work of others and thereby significantly cut the upfront effort involved. We have identified five issures as paramount to the feasibility of such a library, and have concentrated on adapting methodological techniques already familiar in software engineering.

TOWARD A LIBRARY OF FORMAL DESIGNS OF SOFTWARE:

Final Report*

by

Ralph M. Weischedel
(Principal Investigator)

and

Linda Salsburg

*Final report of research sponsored by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under Grant No. AFOSR-78-3539. The United States Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation hereon.

Abstract

Many of the problems of large software systems have been attributed to the design phase of software development, problems such as high maintenance costs, the predominance of software error types traceable to design (rather than to coding), and the high cost of diagnosing and correcting design errors. The most promising approach to these problems is the formal specification of module interfaces, during the design phase, based on the information-hiding principle.

The advantages of formal specifications are as follows: (1) Their precision, lack of ambiguity, and attention to detail should cut down on design errors. (2) They provide informal verification of a hierarchically designed system while it is being designed. (3) Special design validation teams could rigorously verify a design before it is implemented, perhaps with the aid of automated tools for some of the verification. (4) Formal specification enables rigorous specification of the requirements that an embedded computer system must conform to. (5) They combine with the information-hiding principle to enable design of systems that are much easier to modify and maintain.

Yet, the formal specification of modules itself has a serious drawback: the upfront effort in creating them is considerable. Consequently, this research has investigated

the feasibility of a library of formal specifications so that designers could build on the work of others and thereby significantly cut the upfront effort involved. We have identified five issues as paramount to the feasibility of such a library, and have concentrated on adapting methodological techniques already familiar in software engineering.

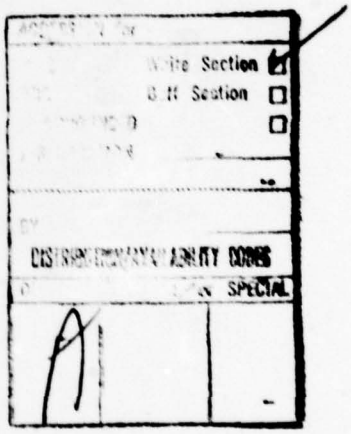


Table of Contents

Appendices

1. Conclusions and Recommendations

This final report is organized as follows: this section briefly summarizes the conclusions and recommendations of the research after a very short exposition of the purpose of the project. Section 2 introduces the research by explaining its motivation and purpose. Section 3 introduces the major issues that we considered. In section 4, we itemize exactly what we did and the analysis techniques used. Detailed results are justified in section 5. The relation of this work to that of others appears in section 6. Several appendices are included. One describes the specification language used in the examples; the remaining four are a small sampling of module specifications written during the course of the project.

1.1 Brief introduction

Large software systems can be defined as systems requiring at least 25 programmers to develop and requiring at least 30,000 lines of source code. Case studies (Boehm, et.al. (1975) and Hamilton and Zeldin (1975)) of such systems have shown that software errors traceable to the design phase of system development can account for as much as two-thirds to three-quarters of all error types, even on generally good software systems, compared with as little as one-quarter to one-third of all error types traceable to the programming or coding effort. The types of errors traceable to the design phase are

predominantly errors in the interface of subsystems or modules.

Design errors and design validation are particularly crucial in embedded computer systems. An embedded computer system is a system developed specifically to function as a component of a larger system.

A design technique to address these problems was developed by Parnas (1972). It concentrates on defining the interface of a module in enough detail that all other systems may use it, but completely independent of the way the module may be implemented. Since programming languages focus attention on how to implement a module, several new languages for formally specifying the interface of modules have been developed in the last few years.

The technique of formal module specifications seems to offer much toward solving many problems including: the high cost of software maintenance, the predominance of design errors, the difficulty and cost of diagnosing and correcting design errors, and the special design and maintenance problems of embedded computer systems. Yet, the creating of formal specifications is very difficult, requiring much upfront effort, and therefore the solution itself is rather costly.

The natural solution is to develop a library of formal specifications of modules. By reusing module specifications, rather than recreating them, the cost of the upfront effort would be dramatically cut and the

problems listed in the preceding paragraph would be solved.

That is the motivation of this research in examining the feasibility of such a library. Several issues are involved in such a library being feasible. To answer each question about the feasibility of a library of formal module specifications, we have given special attention to techniques that are already available in software practice and could be adapted to these particular questions regarding the feasibility of a library, (rather than spending effort on developing yet another language or another methodology). Since formal specifications are so hard to create and demand a way of thinking about design which is rather new to most individuals, our focusing on familiar software techniques that could be adapted to the problems of such a library has the additional benefit of making the library far more practical and usable.

1.2. Summary of conclusions

Three techniques have proven very useful for detecting flaws, omissions, and errors in the formal specifications of module interfaces. One is an automatic tool (Roubine and Robinson, 1976) which checks for syntactic correctness and some simple semantic problems, such as referenced but undefined functions or variables. A second is a walkthrough to check the correctness and completeness of the interface and to criticize the

readability of the specification. (For a library of such specifications, the guidelines we have suggested for documentation would enable each user of the library, in essence, to perform a walkthrough of the specification, informally checking its correctness. Thus, with each use of a module from the library, our confidence in the correctness of the specification can justifiably increase.) The third technique is to write a very rapid implementation of the specified module in a very high-level language. Each technique uncovers flaws or omissions in the specification.

Rigorous design validation before coding even begins would be possible using formal specifications, since an independent verification and validation team could be called in to verify the design. For instance, the designs of two operating systems, (Neumann, et.al., 1977 and Ford Aerospace, 1978) have been proven to maintain certain security properties before implementation began.

Automatic retrieval of module specifications from a library is analogous to the bibliographic retrieval problem. Since that is such a difficult problem involving so many research projects, we concluded not to address that aspect further, but merely suggest the use of techniques such as keyword analysis.

For a given application, it is crucial that only a few prototypes in the library cover the vast majority of variations. For if more than a few prototypes must be

considered, too much of the designers' time will be taken studying the many alternatives. Since the formal specifications are difficult to understand, it is unrealistic to expect that a designer can carefully study more than a handful.

In the examples we have written during the research, we have found that two to four prototypes seem to cover most needs for a particular type of data structure and three prototype text editors seem to cover most varieties of text editors. This was accomplished by following three principles for writing and selecting prototypes for the library.

1) Details that are somewhat arbitrary and which the designer may wish to modify should be localized into subdefinitions so that the modifier need modify the specification in only one place for each detail.

2) Aspects of the module that are not fundamental need not be specified in the library. A whole host of valid command languages may be added on to the logical abilities of any given text editor; therefore, our prototype specifications did not include a command language.

3) The prototype should include as many logically different operations as possible, since the designer can easily delete ones unneeded in a particular environment.

Although the results using the three principles are very promising, we simply have insufficient evidence to judge whether application areas other than data structures and text editors will also be covered by only a handful of prototypes. Further research and experience is definitely needed to answer this question.

A third issue is what to store in the library. Clearly, the specifications of the module interfaces must be stored, for this is prerequisite to the modern design methodology proposed by Parnas and many others to solve the problems of large software systems and embedded computer systems. For any specification, many hierarchical program designs and many programs implementing it are possible. Each will have differing space and time requirements, and therefore are necessary alternatives for differing environments. This could involve several programs for each prototype module interface. Furthermore, the need of a designer to tailor the interface of the module slightly for his/her particular needs, as one would expect to happen normally, requires modifying any selected program implementing the module. Consequently, we have no definite answer as to whether storing the alternative implementations as well is

feasible.

We have identified six items that must be stored with the module specification as documentation. These are short descriptive items designed to enable a designer to quickly identify a handful of interface specifications that are closest to his/her needs.

Since formal module specifications are so hard to understand, a fourth issue is how to make them more understandable. We have identified three reasons for the difficulty in understanding formal specifications.

Programming languages focus on implementation detail; the module interfaces must be specified independent of implementation detail. Consequently, the semantics of these languages is fundamentally different than the languages we are familiar with for programming. Therefore, rather than trying to devise a new language, we have shown how to adapt many of the principles of writing well-structured, easily understandable programs to the task of writing understandable module specifications.

The last issue is how to define the interfaces to be stored in the library so that they may be easily modified for or tailored to the demands of a specific environment. Two principles were discovered for this. One is to localize any detail that is somewhat arbitrary in a definition; then, to modify the detail, only one change need be made. Second, the author of the specification should make a list of the details that he/she

suspects might need modification, so that the author's anticipation of possible changes saves the designer's effort.

1.3 Recommendations

For each module specification written, we quickly implemented a corresponding program in a very high level language as a means of testing the completeness of the interface specified. The rapidity with which we could create this quick-and-dirty implementation suggests that very high level languages might be the means of creating software breadboards. Though the breadboard would be wasteful of memory and time, it could be created quickly to demonstrate the functional characteristics of the system. If a scale factor for the speedup and the economy in memory usage can be estimated for going from the breadboard to a production system, the breadboard would also provide ballpark estimates for the performance of the module.

In this way, the breadboard would provide end users with actual use of the system under design as a means of design validation. Furthermore, we found that a significant portion of the specification language features had a standard translation into the very high level language. Consequently, a software tool could be designed to perform much of the translation from specification language to very high level language automatically.

Therefore, we recommend empirical research in the use of an existing high level language, such as INTERLISP (Teitelman, 1975), for the purpose of rapidly building breadboard systems corresponding to a design. Research in building and experimental use of a semi-automatic tool to translate a significant portion of the specifications into a very high level language also seems to be a high-payoff area for investigation. (The software tool could not be fully automatic since the specification language specifically leaves out implementation detail. So, some coding by a human is always necessary.)

Another area for further work is to specify many more modules from a diverse class of applications. Our results were promising, since a handful of prototypes could cover most needs in the areas of data structures and text editors. However, our experience is simply too limited to project whether this key ingredient to the success of such a library will carry over to other applications.

In the same way, we did not have time to experiment with storing several alternate implementations for each module specification. For this to be feasible, the number of alternate implementations per module should be small. Also any requirements to modify or tailor a prototype module interface to a particular environment must be easily implementable changes in the alternative programs stored with an interface specification. This is

certainly worthy of further research.

Our final recommendation is that a library such as we have proposed should not be begun for actual use until specification languages leave the experimental stage. These languages, being very young, will continue to be enhanced, particularly in the area of specifying the semantics of parallel computing. Liskov (1977) provides a survey of the state of the art in specification languages.

2. Motivation of the Research

The attractiveness of having a library of the products of previous software development is the benefit of building on the work of others. There are several reasons in particular for wanting a library of formal specifications of modules, the key product of the design phase of the software development process. We present these reasons in the following subsections.

2.1 The cost of software and its maintenance

The high cost of software is well-known and need not be belabored; however, the immensity of the problem can be quickly indicated by such facts as the following. It is estimated in Boehm (1977) that the annual cost of software in the United States is $20 billion dollars, or almost 2% of the entire gross national product. In a study for the Air Force, Boehm (1973) reported that the estimated software expenditures of the Air Force for 1972 were three times the hardware expenditures. Even more startling is the projection from the same study that by 1985 the annual software expenditures of the Air Force would be nine times the hardware expenditures.

At the same time, the maintenance costs of software are of deep concern. Large software systems are notoriously difficult to modify in response to changing needs or the discovery of bugs not found until delivery of the product. For instance, OS/360, a major operating system for the IBM 360 series of machines, is one of the

largest software systems ever developed. It is estimated, (Boehm, 1973), that with each new release of the system, 1000 new software errors were introduced.

Insight into the causes of such massive software modification problems is given in the case studies cited in the next section.

2.2 Design errors and the importance of design validation

A major cause of the maintenance problems of software is the design of large software systems, as evidenced in the following case study by TRW. Boehm (1975) reports that the software studied is an example of a generally good system, delivered on time and within budget by TRW. The system consisted of 100,000 source code statements and underwent five modifications during its life cycle, ranging in size from just under 1000 up to 10,000 source code statements changed.

The errors found in making the five modifications were categorized into 224 types. In a summary analysis, the error types were classified as to their origin, either traceable to the design phase (requiring a modification of the detailed design description) or attributable to the coding phase of software development. Error types traceable to the design phase tended to involve interface problems between the modules of the system. "Not only did the number of types of design errors outweigh the coding error types, 64 percent to 36

percent, but also the design errors took the longest by far to detect," (Boehm, et.al., 1975, pp. 125-126). In fact, 54 percent of the error types were not found till integration test or later; of these, 5 out of 6 were design errors. Thus, the overwhelming majority (75%) of the coding error types were found before integration test, but the overwhelming majority (70%) of design error types were undetected until integration test or later. In a related study, it was found that the average time to diagnose and correct design-type errors was about twice that of coding errors.

Similar results have been reported by Hamilton and Zeldin (1976); 73 percent of all errors found during integration tests of the APOLLO project were interface problems.

Clearly then, the design phase of software development is most critical in building large software systems that are both maintainable and reliable. Design validation is therefore crucial before coding begins.

For a number of reasons given in the next section, the formal specification of modules has been suggested by many computer scientists as the solution to these problems.

2.3 The attractiveness of formal module specifications

By a "module," we mean a tightly related group of algorithms and data that provide some computational facility. Because of the aforementioned problems of

maintaining software, the information-hiding principle was suggested by Parnas (1972) and has been widely accepted since then. According to this principle, one should modularize a system by identifying those design decisions which might be reversed due to changing conditions during the life-cycle of the system. For each such design decision, a module is defined such that the result of deciding that design issue is hidden in the module and invisible to processes or people using the module. The information-hiding principle requires that there is a technique for precisely specifying the interface of a module and the visible behavior of the module in all circumstances.

In this way, designers can completely specify the modules of a system such that a programmer team can be assigned to each module and such that each team can work independently without knowing any more about the other modules than the specifications of the interfaces. Since each module depends on only the interfaces of the other modules, any of the design decisions hidden by the modularization based on the information-hiding principle can be reversed easily, for only one module must be modified. Parnas (1976) demonstrates how this technique enables the specification of a family of closely related software systems such that moving to another family of the system is much simpler than would be otherwise possible. Modifications of a system due to changing needs and environment during the software's life cycle is like

moving from one member of the family of systems to another.

Given this, there are several reasons why formal module specifications appear so promising for software problems.

1) Several formal languages have been developed for specifying the interface of a module according to the information-hiding principle. They define as much as is needed to know exactly how to use the module without committing the module to any particular algorithms or data structures for implementing it.

2) Formal specifications state the interface of a module precisely, unambiguously, and completely. Design errors resulting from incomplete, vague, or ambiguous module descriptions should be vastly cut down. The case study data cited in 2.2 testifies to how large a percentage of errors are related to interface definition and how difficult and expensive it is to correct such errors compared to coding errors.

3) As Parnas (1977a) points out,

formal specification of modules seems to be a necessary part of a topdown development or stepwise refinement to define the design of a large software system. For one must specify each assumption of the next lower level before proceeding to that level.

4) By using a formal specification, one can prove properties of the system specified. For instance, Neumann, et.al. (1977) presents proofs that their formally specified design of an operating system guarantees certain security properties of the system. Furthermore, informal proofs of the correctness of the design can proceed at each step of a stepwise refinement as the design process is in progress.

5) With a formal specification, a technical team using machine aids could rigorously validate the design before coding begins.

6) It has been shown (Parnas, 1977b) that using the information-hiding principle to define modules

by formal specifications significantly simplifies the problems of developing and maintaining embedded computer systems. An "embedded computer system" is a system developed specifically to be a component of another system. Such systems arise in Defense Department needs.

2.4 Problems with formal specifications of modules

Though the formal specification of modules seems to offer so much in solving the problems of large software systems, there are problems with using formal specifications. They are very hard to create, and therefore require considerable upfront investment of effort in the software development process. In speaking of this upfront effort, Parnas (1976, p. 7) comments, "The method permits the production of a broader family and the completion of various parts of the system independently, but at a significant cost. It usually pays to apply the method only when one expects the eventual implementation of a wide selection of possible family members."

Each significant modification of a software system during its life cycle corresponds to moving from one member of the family of software systems to another. Consequently, whenever it is likely that significant modifications will become necessary during the software life cycle, a broad family of software systems must be

possible from the design. For embedded computer systems, the likelihood of change throughout the life cycle of the system is particularly acute.

2.5 The need for a library

Thus, the technique of formal module specifications seems to offer much toward solving many problems including: the high cost of software maintenance, the predominance of design errors, the difficulty and cost of diagnosing and correcting design errors, and the special design and maintenance problems of embedded computer systems. Yet, the creating of formal specifications is very difficult, requiring much upfront effort, and therefore the solution itself is rather costly.

The natural solution is to develop a library of formal specifications of modules. By reusing module specifications, rather than recreating them, the cost of the upfront effort would be dramatically cut and the problems listed in the preceding paragraph would be solved.

That is the motivation of this research in examining the feasibility of such a library. The issues involved in such a library being feasible are described in the next section.

A library of formal specifications of modules has an additional benefit for design validation. If the author of a formal specification follows the guidelines for documentation that we have developed during this

research, each reader of a formal specification would be able to informally verify that its specification satisfies the properties the author claims for it. Therefore, with each use of a specification of the library, our confidence in the correctness of the specification will justifiably increase.

To answer each question about the feasibility of a library of formal module specifications, we have given special attention to techniques that are already available in software practice and could be adapted to these particular questions regarding the feasibility of a library, (rather than spending effort on developing yet another language or another methodology). Since formal specifications are so hard to create and demand a way of thinking about design which is rather new to most individuals, our focusing on familiar software techniques that could be adapted to the problems of such a library has the additional benefit of making the library far more practical and usable.

3. Issues

There are five issues that must be resolved to determine the feasibility of a library of formal designs of software modules: how to retrieve items from the library, whether a few prototypes for a given application will cover the vast majority of possibilities for that application, what a library item should consist of, whether the formal specifications will be understandable, and whether they can be written in a way as to be easily modified.

For each of these issues, we describe their significance in this section. Details of the methods used in answering these questions are given in section 4; except for the question of retrieval which is answered in 3.1, all other questions are answered in section 5.

3.1 Retrieval

Clearly, retrieval from the library is a crucial question; yet it is a most difficult one. One class of data base techniques is oriented to storage and retrieval of individual fields from records satisfying particular properties. For instance, a company might have a file of employee records with one record per employee, each record consisting of fields such as name, social security number, address, birthdate, department, starting date with the company, present salary, etc. Each field represents factual information about the person or entity represented by a record. Typical

retrieval for such a data base might include a command such as "List all employees in our marketing department who earn more than $25,000." There are well defined and researched technologies for such data bases, including relational data bases, hierarchical ones, and CODASYL data bases.

The entity in question for our library or data base is a module specification. Unfortunately, what a user of the library would like to know about one is not characterizable by a set of facts, as in the example mentioned above. Rather, a designer wants to know the function, purpose, or mission of a given module and see what the closest match to his/her need is.

This is analogous to the user of a library wanting all books or journal articles related to a specific need or topic. Just as in the case of bibliographic material, there is one dominant question: "Find all module specifications (or books in the bibliographic case) that address the following need (or topic in the case of bibliographic retrieval)."

Consequently, the relevant data base techniques are those for bibliographic retrieval. Given this analysis, we decided rather early in the grant period not to investigate this question further for the simple reason that so many people are studying the problem of bibliographic retrieval that this effort would better be spent on issues solely related to module specification.

For retrieval purposes, we therefore recommend

using keywords to describe the purpose of the module.

3.2 Prototypes

Closely related to the problem of retrieval is the following crucial question: for any particular application need, will there be a small number of prototypes which cover most of the variations possible? The ideal would be that a handful of prototypes for a given need would cover the major possibilities, so that the designer can quickly ascertain which, if any, fits best. If there are many prototypes necessary for each application, then the time spent analyzing each one will make using the library prohibitive.

3.3 What to store

The term "design" covers many aspects of software development, ranging from (a) partitioning a system into modules and precisely defining the interface of each module, to (b) defining the detailed control structures of a procedure or algorithm using a program design language. Of the products which result from the various design activities, which should be stored? Could all be meaningfully stored? Could programs implementing a particular design be stored as well?

Furthermore, what kinds of documentation can be provided with each of the prototype module designs so that a user of the library can quickly discard ones that do not fit his/her needs and concentrate on only a few

for in-depth study?

3.4 Understandability

The formal designs stored must be understandable. Even if the previous three issues are satisfactorily solved, a library of formal designs of modules will be useless unless the formal specifications are understandable. The user of the library must be able to understand the specifications in order to decide whether the design meets his/her needs and to successfully integrate it into the rest of the system of which the selected module is a part. The module must be integrated into the system both at the design level and at the implementation level.

Unfortunately, understandability of formal design specifications is the overwhelming problem with the specification languages available. Consequently, we have concentrated most on this one issue.

3.5 Modifiability

Given that the most appropriate specification is found, it may not be a perfect match to the designer's needs. In that case, the more easily the specification can be modified to suit those needs exactly, the better. What techniques are there to write specifications which are easily tailored to meet a slightly different need?

4. Method used in studying the five questions

Our method has been to specify as many modules as possible to build up experience upon which to suggest answers to the five questions. Study of these examples provides the basis of our conclusions. The examples fell into two general classes: data structures and text editors.

Commonly used data structures were chosen for several reasons. First, they are crucial in virtually all programming applications; hence, they would have to be a part of the proposed library. Second, the overwhelming majority of module specifications appearing in the literature are just those data structures. Third, they are simple enough that we could specify several during the grant period; because of their simplicity they would be easily understood by anyone unfamiliar with the formal language SPECIAL, (Roubine and Robinson, 1976), which we used for our specifications.

Text editors were chosen because they represent a realistically complex application without being intractable in the time available.

For each application considered, we tried to conceive of all meaningful variations that would arise within that application. Any significant difference in the functional capabilities of the module was considered a meaningful variation.

For a last-in-first-out storage structure, we specified two modules: a stack from which one could

read values from only the last item stored, and a stack which additionally had a movable pointer into the stack. Any value designated by the pointer could be read, but not modified. (This variation is useful in interactive debugging aids to recursive programming languages such as Algol, Pascal, APL, and Lisp, for the programmer can then examine the sequence of procedure calls when an error occurs.)

For a first-in-first-out storage structure, we specified four variations: a normal queue where reading occurs at only the front position of the sequence, a priority queue, and two character streams. In a priority queue, the first entered of the largest values (highest priority) is read or removed from the sequence before any others; no other values can be read. Character streams are very useful abstractions of input and output; for they enable the details of synchronizing input/output operations to be hidden in the module rather than forcing all programs to be aware of the means of synchronization. The basic operation of one of the character streams is character oriented; for the other version, the basic operation was oriented to blocks of characters or lines.

The ubiquitous tree was another application which we specified. Two variations, the binary tree and general tree, seem to cover all logical variations. Though we specified a binary tree, we did not get to the general tree. One might wonder why a threaded tree is not

a third logical variation. The reason is simple: a threaded tree is an implementation of a fast means of performing tree traversal, an operation on trees. Consequently, the specification of binary trees includes three operations for inorder, postorder, and preorder traversal; the details of how the traversal is performed (perhaps via thread links) is not properly part of the specification, for it would violate the information-hiding principle.

For text editors, we found three logical alternatives. One has operations oriented to adding, deleting, or moving characters; a small, basic subset of the operations of TECO, a popular editor on the DEC series of machines, was specified. Another module specified has operations oriented to lines and line numbers; a basic subset of the commands of SOS, another popular editor on DEC machines, was specified. The third example is an editor whose operations are oriented to moving a cursor on a CRT screen and editing on the screen; NED (Bilofsky, 1977), an editor running under the UNIX operating system, was used as a pattern of the functional capabilities of such systems.

For each of the modules specified, several techniques were used to check the correctness of the specifications. (By "correctness" of a specification, we mean that it specifies exactly the properties desired.) One was to use an automatic tool, the specification handler for SPECIAL (Roubine and Robinson, 1976). This checks

for syntactic correctness of a specification. Since SPECIAL is a strongly typed language (every expression has a type such as INTEGER, BOOLEAN, etc.), the system can check that the types of each expression always agree with what they should be. (Due to an oversight, the priority queue was never run through the specification handler.)

A second check was for someone other than the author to scrutinize the specification for anything that seemed to disagree with the author's claims; this is like a walkthrough.

Third, a quick implementation corresponding to the prototype was made in a very high level language (LISP). (An implementation of the third text editor was not started due to the termination date of the grant; all other modules were implemented.) This provided a concrete system for testing the functional capabilities of the module; it often uncovered specification errors not found by the previous two methods. The implementations took remarkably little time; by using a very high level language one could trade the performance characteristics of the implementation for vastly reduced programmer time. Consequently, the final module we checked in this way took only two to three days for one programmer to implement, even though it was the character-oriented text editor, a significant, realistically complex module. Furthermore, we found significant regularity in the implementation of most types of expressions

specified in SPECIAL. This suggests that a significant proportion of each implementation could be done automatically by a software tool.

5. Detailed results

In section 3, we isolated five questions about the feasibility of a library of formal designs of software modules. Section 4 presented the empirical study performed to answer the questions. One question, how to retrieve items from such a library, was answered in section 3. Our results on the remaining questions are detailed here.

5.1 Prototypes

For any particular application, will a handful of prototypes cover most of the possible modules to meet that need? If more than a handful are needed for any particular need, the designer will have to spend too much time analyzing each one to find the best for his/her needs.

Our experience in writing many specifications is very encouraging; however, it is simply insufficient to answer the question conclusively.

In the area of data structures, two to four prototypes seem sufficient to cover the various significant alternatives. For instance, for a stack, or last-in-first-out storage structure, the two alternatives were a stack allowing retrieval of values only at one place (the "top") and a stack allowing retrieval of values from any position designated by a movable pointer.

Although certain details of each specification might be changed by the designer, details such as

whether trying to read from an empty stack causes an error or not, are not significant enough to warrant storing additional prototypes. Rather, the author of the specification in the library can include in the documentation notes to the readers (designers) about details that might be modified. In fact, our documentation for each example includes that.

In addition to the area of data structures, our examples dealt with medium-sized software tools, such as text editors. (By software tools, we mean software that serves as an aid in writing other software.) Though there was only enough time to study the one application area of text editors, our experience was very encouraging. Thus, we cannot state conclusively that for all classes of software tools, a handful of prototypes will largely cover all variations, but we suspect that this will hold true.

For instance, there are a vast variety of text editors available. However, they seem to be classifiable into three categories. The first category is text editors whose operations are oriented to insertion and deletion of characters relative to a current position. Line numbers may not be used at all; rather, one moves the pointer defining the current position to indicate where editing is to occur. TECO (Digital Equipment Corporation, 1972), a DEC product, is an example of this class. The second category is text editors whose operations have a sequence of line numbers as an argument and

which are therefore oriented to operations on lines. SOS (National Institute of Health, 1977), another DEC product, is a particularly rich example of this class in that the user may leave the line-oriented mode and move to a character-oriented mode on a range of lines. The third category is text editors whose operations are based on viewing a CRT screen as a window into the file, where editing can take place. Editing occurs within the window relative to the position of a cursor. NED (Bilofsky, 1976), is an example.

Almost all general-purpose text editors can be classified into one of the three categories. (One exception is the special-purpose editors created for editing LISP programs (Sandewall, 1978).) However, that does not prove that three prototypes will cover most applications, unless one follows these principles in creating the specifications for the library.

1) Certain details of a specification will vary given the environment in which the module is to be used. Select a consistent set of decisions for the specification in the library. As much as is possible, localize each such detailed decision in a subdefinition of the specification; in this way, a user of the library can modify the arbitrarily chosen detail just by changing the subdefinition. (One of the extremely useful features of

SPECIAL is that the language provides many mechanisms for making such subdefinitions.) In the documentation, all details which are arguable must be clearly indicated.

2) In specifying a new entry for the library, avoid differences that are not fundamental to the logical, functional capabilities of the module. Those differences would multiply the number of entries for a given application in the library without adding any new abilities. For instance, in specifying text editors, we did not define a user command language, for there are many legitimate syntactic variations, each of which will be of varying value to different user communities.

3) The author of a new entry for the library should specify as many logically different, primitive operations as can be imagined. A designer, after selecting a specification from the library, can delete operations from the interface not needed in his/her environment. This, of course, assumes that each operation at the interface was defined using information-hiding as stated in principle (1). For instance,

our specification of a line-oriented editor was patterned after the functional capabilities of SOS. The operations corresponding to the alter mode, where the user can modify a range of lines using character-oriented operations relative to a pointer, can easily be removed.

In conclusion, our experience is too limited to determine whether for all applications, only a handful of prototype specifications will cover almost all variations in modules for the particular application. However, the three principles above did enable us to specify a handful of prototypes successfully covering most anticipated needs in the application area of text editors and in the class of applications of data structures. That fact is quite encouraging.

5.2 What to store

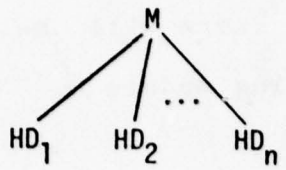
For any given module specification to be added to the library, what should be stored? The functional specification? A hierarchical design for implementing the module? A complete program? Furthermore, what kinds of documentation are necessary?

Clearly, the abstract, formal specification of the interface of the module must be stored, for this is central to hierarchical design of a large software system. The specification of the module's interface defines each functional capability of the module, specifying the

arguments of each function call, and the effect of any operation or function at the interface of the module. Of course, this information is exactly what is necessary for the independent implementation and use of modules according to the information-hiding principle (Parnas, 1972). Furthermore, the specification of the module interfaces is necessary for embedded computer systems.

Thus, the formal specification of the module's interface must be included, and we have therefore concentrated our research on this. Yet, could one store an implementation as well? In general, any module of significant size, when implemented would itself be hierarchically broken down into modules that together implement it. Furthermore, there would normally be several alternate ways of hierarchically defining modules to implement the one of interest. Thus, once the designer has selected a module specification meeting his/her needs, a second selection must be made from several alternate hierarchical designs implementing the module. There is no technical problem to being able to specify the hierarchy, for SPECIAL, as with other specification languages, permits the formal definition of a hierarchy implementing the functional capabilities of a module as well as the formal definition of those functional capabilities.

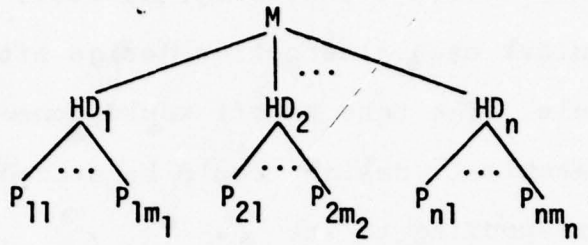
The diagram below summarizes this. Associated with each module specification M, there would be a few hierarchical designs HD for implementing it.



It is possible technically; however, the designer must consider each alternative design after selecting a given module. The true payoff would come if not only the hierarchical design could be stored but also programs corresponding to it.

Again, even for a single hierarchical design, there will in general be several significant, alternate programs implementing it. Given a module's specification, the differing hierarchical designs for it and the various programs implementing each design will offer various tradeoffs in computing time required, memory used, and other computer resources required. In other words, the performance and computational needs of the various programs make the alternatives meaningfully different. As an example of this, even for a task as simple as sorting a sequence of items, there are several competing algorithms, such as quicksort, heapsort, mergesort, bubblesort, and insertion sort. Each has advantages and disadvantages due to differing performance and memory requirements. Given the environment, the tradeoffs among the algorithms can be resolved to select a particular program. Thus, we have the situation described in the figure below. For each module M, there will be a few associated hierarchical designs HD. For each

hierarchical design, there will be a few programs P associated implementing module M.



It is technically feasible to include with the functional specification of the module's interface various hierarchical designs and corresponding programs implementing the module. However, since the number of implementations that may need to be stored for any module could be relatively large, developing a reasonably complete library including implementations as well could take many years. However, the library would be of great value to designers just with the module interface specifications as the alternative programs for each module are added slowly.

Furthermore, even with a complete library, some reprogramming would always be necessary. The reason is that the module specification M itself may oftentimes need slight modification due to varying environments in which the module is to be used. This will require modification of one of the library's programs

implementing it as well. For this to be feasible, each program stored would have to be very well structured and very well documented.

To repeat, in this study we have concentrated exclusively on the feasibility of building a library of module specifications M, defining the interface of a module.

What documentation must be stored with these module specifications M? Detailed documentation must be stored in the formal specification itself so that a comment is with the expression it is describing. Our thoughts on such commenting appear in section 5.3. Additionally, a short high-level description is crucial to the effectiveness of the library so that the designer may quickly ascertain whether a module specification is relevant to his/her needs. The designer must be able to quickly eliminate ones that are not close to the need so that he/she can focus attention on two or three that are the most promising. Otherwise, the library would bog designers down on decisions that should be made quickly.

We have found six types of information that seem valuable for quickly deciding on the relevance of a module specification. One is a list of keywords describing the purpose of a module. A second item is a description of the purpose of the module and the kinds of needs it fills. A third item is a summary description of each class of functional capabilities the system has.

Fourth, the kinds of decisions not made by the module are described; these are implementation decisions which would have to be made after selecting a module, when the coding phase begins. These decisions, which are left open by the specification of the module's interface, are a direct measure of how easily the software can be modified during its life cycle. This is because the decision is made at implementation rather than at design, and because the result of the decision is isolated in one module. Therefore, each decision is easily reversed during the system's life cycle by modifying only the one module (rather than having to modify many). For instance, in specifying a text editor, we would write in this section, (which we have called "Information Hidden"), that a decision to use array storage, linked lists, or other alternatives to store the file being edited, would have to be made when programming of this module begins.

Fifth, specific references, such as texts and journals, if available, should be given describing various implementations, algorithms, and analyses of them for use when programming begins. And last, the author of the specification must include all ways that he/she anticipates that the specification might require modification to tailor it to specific needs. Of course, some details of the specification will be rather arbitrary; these must be pointed out so that one specification can serve for many slight variations.

5.3 Understandability

The formal specification of a module M must be understandable if it is to achieve its purpose, for it acts as a contract between designers and programming team, stating exactly what the programming team's product must do (Parnas, 1977a). If a contract cannot be understood, it serves no purpose. However, the understandability of formal specifications in a library becomes even more crucial, for if the designer cannot understand the alternative specifications, how can an intelligent choice be made among the alternatives? Consequently, our research has concentrated most on this single question.

Our results are in two parts: reasons why formal specifications have been difficult to understand and specific suggestions on making them understandable. Researchers in the area of formal module specifications and abstract data types generally agree that they are difficult to understand, though the degree of difficulty is argued.

We have found several reasons for the difficulty of understanding formal specifications, particularly as compared to natural language specifications.

i) Natural language specifications usually do not contain the detail that formal specifications do. Though this is a major reason that natural language is

inappropriate for module specification, the necessary, added detail in a formal specification makes them harder to understand. Liskov (1977) agrees, stating on p. 13-5, "Rigorous informal specifications are probably just as difficult to construct as formal ones; informal specifications appear easier to construct because they are usually incomplete."

ii) Natural language is used by each person daily; thus, a natural language specification has myriads of concepts already defined and familiar to us to succinctly state what a module does. For example, the concepts of a sorted sequence, a pointer, and a line of text, are all well-known and are referred to without further explanation. (Yet, the advantage this gives to understandability is simultaneously a serious drawback to natural language specifications, since the notion raised in each person's mind need not be the same.) Because formal specifications are so new, there is no corresponding body of defined concepts which have been taught us and which have been frequently used by us. Therefore, concepts such as sorted order must be defined in the

specification, thus adding to what must be understood in the specification. Of course, a library of such definitions would provide a body of past experience to study and use just as in natural language. The formal definitions are unambiguous and precise; therefore the definitions do not allow multiple, valid interpretations as in natural language.

iii) Formal specifications of modules are to be abstract and cover many possible implementations. Not only does the abstractness itself contribute to greater difficulty in understandability, but the semantics of the abstract languages is necessarily quite different than the semantics of programming languages, since programming languages are designed to define implementation detail. Thus, the focus of attention of specification languages is of necessity quite different from programming languages. Therefore the semantics of specification languages is quite different than programmers are used to.

The following specific suggestions for making formal specifications understandable have come out of our

research. There are two alternate approaches one could take to make formal specifications more understandable. One would be to abandon the present specification languages and try to formulate one whose semantics is like that of standard programming languages. Yet, as (iii) states, that would fail since specification languages, because of their purpose, must have a semantics different than programming languages. A second approach is to adapt the techniques for organizing clear, understandable program code to the problem of writing understandable formal specifications; this is the approach we have taken.

Therefore, the suggestions are familiar in programming. Our suggestions, rather than being platitudes, such as "Use comments," are specific guidelines, such as when to include a comment and what to write. Only a summary of the guidelines are presented in this section. Fully specified modules using these guidelines are presented in Appendices II through V and also in Weischedel (1979).

The test of these guidelines is whether the module specifications seem understandable to the reader given a reasonable amount of study and some familiarity with the specification language SPECIAL. (A brief description of the major features of SPECIAL is presented in Appendix I). Appendix II specifies a stack; appendix III, a queue, and appendix IV a binary tree. Appendix V is one of the three text editors specified. The examples of a

stack and a queue are intended to provide short, simple cases of familiar concepts to aid becoming familiar with SPECIAL.

Our guidelines are as follows:

1) Since the specification languages, being abstract and nonprocedural, have a semantics very different from that of most programming languages, one should choose a specification language whose semantics draws on concepts common in programming. For instance, one of the strong points of SPECIAL is the familiarity of many of its primitive concepts, including functions, arguments of a function, exception conditions, side effects, declarations, records, sets, vectors, reals, integers, and characters.

2) English documentation should be painstakingly constructed with the following principles in mind:

> a) A high level description of the purpose of a module which is formally specified and a description of each function available at the interface of the module provides a general notion or conceptualization for understanding the formal specification. Though that description will be vague, incomplete, and ambiguous, it conveys a toplevel view around which the complete, unambiguous, precise formal description can crystallize into understanding in

the reader's mind. Of course, the formal specification alone is the arbiter of all questions about the module. Such a high level description is essential documentation for management personnel.

b) In general, formal specifications will consist of a large number of formulas, some of which can be very long. (Consider the examples in the appendices and in Weischedel (1979).) Each formula should be documented, preferably with the documentation intermixed in the formulas, so that the purpose and implications of each subformula are made obvious. The reason is simple: the author of the specification when writing down a long formula had specific reasons for writing each subformula. Those reasons or implications should be succinctly stated. The principle for deciding whether to include a comment for a particular subformula is whether its purpose and implications would be obvious to the average reader without a comment. To illustrate this, figure 1 is given as a definition of a function sort which

returns a sequence b in ascending order corresponding to the input sequence a. Comments are preceded by a dollar sign and are enclosed within parentheses. Reserved words of SPECIAL are typed in upper case. The comments generated by following this principle probably enable most readers to understand and verify the specification even without knowing the detailed syntax or semantics of the specification language.

c) Not only will following (a) and (b) make formal specifications understandable, but also following them gives each reader of the specification the ability to verify that every aspect and subformula of it corresponds to the author's intent. This is an informal means of design validation; for a library of such specifications, the means is very powerful since more and more designers will be reading and verifying a specification as time goes by.

```
VFUN sort (VECTOR_OF REAL a) -> VECTOR_OF REAL b;
  $(Given a sequence of real numbers a, a sorted
    sequence of the same numbers is returned as b in
    ascending order.)


    DERIVATION
      SOME VECTOR_OF REAL b|
        $(b is the permutation of a)
        LENGTH(b)=LENGTH(a)  AND
        (FORALL REAL x|(EXISTS INTEGER i :x=a[i|)
              $(For every value x of the original sequence,
                the following is true)
            :CARDINALITY ({INTEGER j|a[j]=x})
            =CARDINALITY ({INTEGER k|b[k]=x}))
              $(The same number of copies of x are in the
                original sequence a as in the resultant
                sequence b)
        AND $(b is in ascending order)
              (FORALL INTEGER i|(1<=i AND i<LENGTH(b))
                :b[i]<=b[i+1]);
```

Figure 1

(The reader may be tempted to conclude that our guidelines on documentation are obvious. The two most significant, most impressive uses of formal specifications in designing large systems are Neumann, et.al. (1977) and Ford Aerospace (1978), both specifying secure operating system designs. Neither uses comments to the degree we have advocated, and we feel that both would be considerably more understandable if comments were added using our guidelines. Some computer scientists, in fact, would disagree with our guidelines for using comments, feeling that such comments bias the interpretation of the specification or weaken it somehow. For instance, one referee of an early version of Weischedel (1979) wrote, "The English descriptions play too central a role." Clearly, then, our guidelines are not obvious. We have concluded that without these guidelines, formal specifications of large systems are almost impossible to understand. Of course, they do suggest an interpretation, one that is critical to understanding a formal specification and which is valuable for informal verification that it specifies what the author intended. Without such documentation there is just as much chance for misinterpretation of the specification, plus the added drawback that they would not be understandable. The depth of documentation espoused here and the practical principle for deciding whether to include a comment is a practical contribution of our research.)

3) There are several styles of writing even simple

STRUCT_OF (INTEGER name1; CHAR name2) defines records

formal specifications. Sometimes a simple composition of the primitive elements of the language suffices to define a concept. Recursive definitions can provide short, easy to understand specifications. Another alternative is to use the English definition of the object as a pattern for a formal definition; this often leads to definitions in terms of sets and operations on them. In writing the specifications for our library, we have found that the style that seems clearest depends on the item being defined. Understandability of the formal specifications should be an explicit goal of the design phase. Since there are several styles of definition that may be used, walkthroughs were used by us to check not only the correctness of a specification but also its understandability. For the three text editors we have specified, walkthroughs proved very valuable in criticizing both the formal definitions and the documentation. Understandability is not easily attained. For each of the text editors, after completely specifying one, we were able to conceive of a much more understandable, but functionally equivalent specification by significantly changing the style of the definitions. We have concluded that entries for a library of formal module specifications should be prepared with so much care for understandability that casting away the first attempt at a specification to create a more understandable one is not frowned upon. Writing clear nonfiction receives that much attention; so should

creating a specification to reside in a library of module specifications for repeated use.

4) In (ii) it was stated that one reason that formal specifications are harder to understand than English ones if the abundance of concepts already defined in English, but which have yet to be defined in formal languages. Therefore, in English specifications, the concepts may simply be referred to whereas in formal specifications each concept must be defined. This leads to three natural conclusions: (a) a library will add to the understandability of formal specifications as the definitions in the library become familiar to designers, (b) until such a library is built up, module specifications will be rather complex as all concepts must be explicitly defined in a top-down way, and (c) the specification language should provide a rich set of primitive objects and operations on them to alleviate the lack of concepts previously defined. The example of figure 1 illustrates this; the definition would be much shorter and clearer if a concept permutation(a,b) were already defined in the language or in the library. For that matter, the concept of a sorted sequence is so common, that its precise definition should be primitive to the specification language.

While we have not invented "new" ideas for understandability, we have demonstrated that formal specifications can be made far more understandable just by

using many of the ideas espoused in structured programming. The reader is invited to check this claim by comparing our specifications in the appendices and in Weischedel (1979) with those of other authors.

5.4 Modifiability

Can the prototype in the library be made modifiable enough so that minor changes can be made to tailor a specification to a specific environment? There are two simple techniques that we suggest which will make specifications in the library rather modifiable.

The first is to structure the definitions of each function at the interface of a module so that each particular detail that might need modification is localized in only one subdefinition. Then only one subdefinition need be changed rather than the changes being spread throughout the specification. (This is just the information-hiding principle being applied to writing definitions in a specification language rather than its original application as a criterion for decomposing large systems into modules.) One of the nice features of SPECIAL is that several mechanisms are provided in the language for making subdefinitions. In addition to the ability to reference functions at the interface of other modules, one can create subdefinitions within a single module purely to localize information that might be changed in the module's definition. One method is the DEFINITIONS feature for defining particular

subformulas of significance. Another is the TYPES feature for defining a complex new data type used as arguments to functions.

A particularly good example of this appears in Weischedel (1979) where a formal specification is given for an unusually sophisticated pattern matching feature for searching a file. In applying the principle, the toplevel specification of the search feature precisely defined the lines that would be found and that those lines would be returned in order of appearance in the file. The syntax of the pattern language, and therefore what it meant for the search procedure to detect an illegal pattern, was defined in a subdefinition lower than the toplevel. The semantics of the pattern language was also localized in a subdefinition. Therefore, both the syntax and the semantics of the pattern language for the search could easily be modified for different user environments.

The second idea was mentioned earlier. Namely, the author of the specification can anticipate many potential modifications merely by analyzing what details of the specification are somewhat arbitrary (in that the details will vary depending on the environment in which the module is to be used). The author can list each detail which might need modification as an integeral part of the documentation. As an example, consider a queue. In a sequential programming environment, an attempt to remove an item from an empty queue is an

error. However, in a multiprocessing environment, such a condition is merely a signal that the process requesting information should be suspended until some other process adds something to the queue.

These two simple principles will make the prototypes in the library rather modifiable.

6. Related Work

There are several areas of related work. The first, of course, is research in a design methodology using the information-hiding principle. Parnas (1976) demonstrates how modularization of the type assumed here can be used to define a family of closely related software systems, thus making maintenance and modification of software during their life cycle much easier. Parnas (1977b) shows the relation of formal specifications of modules to the special problems of embedded computer systems. Neumann (1977) and Ford Aerospace (1978) are the most significant systems designed to date using formal specification; both are operating systems whose designs have been formally proved to maintain security.

A second, related area is research in specification languages to support defining a module's interface independent of its implementation. This is frequently referred to as abstract data types in the literature. Liskov and Berzins (1977) provide a survey of this research. Ambler, et.al. (1977) presents a specification language which is very similar in semantics to the one we have used. Algebraic languages have the same purpose, though having a quite different semantics. Examples are presented in Guttag, et.al. (1978) and Parnas (1977b). Yet, a third class of specification languages is represented in the ideas of Balzer and Goldman (1979).

A third related area of work is the National Software Works (NSW) reported in Crocker (1975). NSW is a library of software tools available on the ARPA net. A software tool is a software system used in the development of other software; examples include editors, compilers, interactive debugging aids, simulators, document formatters, program verifiers, and automatic test case generators. The purpose of NSW is to enable programmers to use software tools via the ARPA network, tools which would otherwise be unavailable to them. The reason software tools would be unavailable otherwise is threefold: (a) the traditional dependence of software on a particular computer and operating system, (b) the lack of facilities, such as sufficient memory or interactive systems, at the geographical site of the programmer, and (c) the expense of obtaining proprietary software. Thus, it shares the notion of a library to aid software development; yet, its purpose is quite different. Its purpose is to make the execution of software tools available to programmers. The purpose of a library as we have envisioned is to make the formal specifications of module interfaces available to designers for reuse in developing new software. Formal specifications from such a library would enable disciplined design of large systems and rigorous design validation by a validation team.

Bibliography

Ambler, Allen L., Donald I. Good, James C. Browne, Wilhelm F. Burger, Richard M. Cohen, Charles G. Hoch, and Robert E. Wells, "GYPSY: A Language for Specification and Implementation of Verifiable Programs," Proceedings of the ACM Conference on Language Design for Reliable Software, SIGPLAN Notices, 12, No. 3, March, 1977.

Balzer, Robert and Neil Goldman, "Principles of Good Software Specification and their Implications for Specification Languages," Conference on Specifications of Reliable Software, Cambridge, MA, 1979.

Bilofsky, Walter, "The CRT Text Editor NED - Introduction and Reference Manual," Technical Report R-2176-ARPA, ARPA Order No. 189-1, Rand Corporation, Santa Monica, CA, 1977.

Boehm, Barry W., "Software and Its Impact: A Quantitative Assessment," Datamation, pp. 48-59, May, 1973.

Boehm, B. W., R. K. McClean, and D. B. Ufrig, "Some Experience with Automated Aids to the Design of Large-Scale Reliable Software," IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, pp. 125-133, March, 1975.

Boehm, Barry W., "Software Engineering: R & D Trends and Defense Needs," Proceedings of the Conference on Research Directions in Software Technology, Peter Wegner, Jack Dennis, Michael Hammer, and Daniel Teichroew (eds.), 1977.

Crocker, Stephen D., "The National Software Works: A New Method for Providing Software Development Tools Using the ARPANET," Meeting on 20 Years of Computer Science, Instituto di Elaborazione della Informazione del CNR, Pisa, June 16-19, 1975.

Digital Equipment Corporation,, "DEC System 10: Introduction to TECO (Text Editor and Corrector), Order No. DEC-10-UTECA-A-D, Digital Equipment Corporation, Maynard, MA, 1972.

Ford Aerospace, "Secure Minicomputer Operating System (KSOS): Computer Program Development Specifications (Type B-5)," Technical Report No. WDL-TR7932, Ford Aerospace & Communications Corporation, Palo Alto, CA, 1978.

Guttag, John V., Ellis Horowitz, and David R. Musser, "Abstract Data Types and Software Validation," CACM, 21, No. 12, 1048-1063, 1978.

Hamilton, M. and S. Zeldin, "Higher Order Software - A

Methodology for Defining Software, IEEE Trans. on Software Engineering, SE-2, No. 1, 9-32, 1976.

Liskov, Barbara H. and Valdis Berzins, "An Appraisal of Program Specifications," Proceedings of the Conference on Research Directions in Software Technology, Peter Wegner, Jack Dennis, Michael Hammer, and Daniel Teichroew (eds.), 1977.

National Institutes of Health, "SOS (An Advanced Line-Oriented Text Editor) User's Guide," Comp. Center Branch, Div. of Computer Research & Technology, National Institutes of Health, Bethesda, MD, 1977.

Neumann, Peter G., Robert S. Boyer, Richard T. Feiertag, Karl N. Levitt, and Lawrence Robinson, "A Provably Secure Operating System: The System, Its Applications, and Proofs," SRI Project 4332, Final Report, Stanford Research Institute, Menlo Park, CA, February, 1977.

Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," CACM, Vol. 15, No. 12, pp. 1053-1058, December, 1972.

Parnas, D. L., "On the Design and Development of Program Families," IEEE Transactions on Software Engineering, Vol. SE-2, No. 1, pp. 1-8, March, 1976.

Parnas, David L., "The Use of Precise Specifications in the Development of Software," Information Processing-77, B. Gilchrist, (ed.), North-Holland Publishing Company, New York, 1977a.

Parnas, David L., "Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems," NRL Report 8047, Naval Research Laboratory, Washington, D.C., June 1977b.

Roubine, Olivier and Lawrence Robinson, SPECIAL Reference Manual, Technical Report CSG-45, Standard Research Institute, Menlo Park, CA, August, 1976.

Sandewall, E., "Programming in the Interactive Environment: The LISP Experience," Computing Surveys 10, No. 1, 35-72, 1978.

Teitelman, Warren, "Interlisp Reference Manual," Xerox Palo Alto Research Center, Palo Alto, CA, 1975.

Weischedel, Ralph M., "A Tutorial Example on Writing Understandable Formal Specifications of Software Modules: An Extended Abstract," Proceedings of Micro-Delcon, IEEE Publication No. 79CH1426-6C, IEEE, Inc., New York, NY, 1979.

Appendix I

This section is written to help the reader understand a specification written in SPECIAL (SPECIfication and Assertion Language) Roubine and Robinson, (1976). The discussion of SPECIAL will not encompass all of the language's features. The explanation is intended as a guide in understanding the examples which accompany this report.

The design of a large software system requires breaking the system into component parts called modules. Parnas (1972) describes a criterion called the information-hiding principle for deciding how to decompose a system into modules. According to this design principle, any decision which may need to be modified during the system's life cycle should be localized in a single module. In that way, if the results of any decision implemented should be changed during the system's life cycle, only the one corresponding module must be changed, thereby greatly simplifying maintenance.

To achieve this, a designer must specify the interface of the module completely enough that any other module or any user program can call the module and know precisely <u>what</u> the module will do. However, the interface must be specified in such a way that other modules or programs do not depend whatsoever on <u>how</u> the module performs the task requested. SPECIAL is a language designed specifically for these goals. To implement such a module, one writes a set of closely related

programs and data structures conforming to the interface specified.

One can view such a module as an abstract machine, abstract in the sense that its implementation is unknown to other modules, and abstract in the sense that it is most probably implemented in software rather than hardware. SPECIAL is based on the analogy of an abstract machine. A real machine, such as a particular computer, is always in some state, represented by the contents of the CPU registers and the contents of main memory. Similarly, a module or abstract machine is always in some state, represented by a group of values. One can have any portion of a computer's state printed, for instance, output of the value of any main memory location, of a general register, or of the program counter. By the same token, SPECIAL lets a designer specify functions called VFUNs which output or return the value of any of the group of values forming its present state. A computer also has operations which, given the current state, change the state; for instance, an ADD instruction on the PDP-11 takes the sum of the contents of two memory locations and stores it back in one of the memory locations, thereby changing the state of the machine. In the same way, a SPECIAL module has a number of operations (called OFUNs) at its interface which change the state of the abstract machine or module, given its current state, by changing some of the values comprising to current state.

The heart of a module specification in SPECIAL, therefore, is a list of functions available at the interface. There are several other parts to a specification, which are not as central as the list of functions.

Each specification describes some module of a software system. SPECIAL's reserved words (such as MODULE, TYPES, DEFINITIONS and IS) are denoted here by capital letters. A module is preceded and followed by the words MODULE and END_MODULE. The name of the particular module immediately follows the word MODULE. The remainder of the specification is broken up into paragraphs or sections, each beginning with a reserved word such as TYPES, DECLARATIONS, PARAMETERS, DEFINITIONS and FUNCTIONS. The paragraphs are optional. Therefore, the structure of a module is as follows:

```
MODULE <module_name>
     TYPES
       .
       .
       .
     DECLARATIONS
       .
       .
       .
     PARAMETERS
       .
       .
       .
     DEFINITIONS
       .
       .
       .
     FUNCTIONS
       .
       .
       .
END_MODULE
```

Comments may appear throughout each module. A

comment is preceded by a "$" and is delineated by matching left and right parentheses. Comments have no effect on the meaning of the specification but only serve as an aid to understanding.

The best way to read a module is to study the PARAMETERS section first followed by the FUNCTIONS. DEFINITIONS, TYPES and DECLARATIONS paragraphs should be referenced as needed. Comments at the beginning of a specification will suggest an alternative plan when it seems appropriate.

The specification language defines objects and operations on them. Associated with each object is a name and a type. The type of an object defines a set of legal values that the object may have.

Primitive types are disjoint sets of primitive values which may be either predefined or designators. Predefined types include BOOLEAN, INTEGER, REAL and CHAR (character). A designator type is a class of names for user-defined objects. This new class of objects is defined by a particular module and can only be created or modified using functions in that module.

New types can be formed by applying type-constructors to the primitive types. The type-constructor VECTOR_OF creates a type which is a vector (or randomly accessible sequence) of objects for a given type. For example, the type VECTOR_OF CHAR is the set of all possible sequences of characters. Another type-constructor is STRUCT_OF, which defines a record.

3) There are several styles of writing even simple

STRUCT_OF (INTEGER name1; CHAR name2) defines records consisting of two fields; the first contains an integer and may be accessed by the name name1. The second field contains a character and may be accessed by using name2. SET_OF creates a type which is a set of objects of a given type.

The purposes of the TYPES paragraph are explained with the following example.

```
TYPES

graphs:  DESIGNATOR;
sequence:  VECTOR_OF INTEGER;
```

As a result of this TYPES paragraph, a new primitive data type known as "graphs" is introduced. Since it is a DESIGNATOR, its values come from a unique set of names. In addition, the name "sequence" is associated with the type specification VECTOR_OF INTEGER.

The objects defined by the specification language have a scope as well as a name and a type. Once an object has a name and a type associated with it, the scope dictates which parts of the module may reference that name. The DECLARATIONS paragraph is one way to associate a name with a data type. Furthermore, the scope of that name will be global, i.e. its type will be the same throughout the entire module.

```
DECLARATIONS

graphs digraph, cyclic;
INTEGER i ;
```

"Digraph" and "cyclic" are both objects of type "graphs"

and "i" is now declared as an INTEGER.

The PARAMETERS paragraph is used to give an object a value when the module is initialized. The value is a constant which would receive a particular value only after implementation of the module, when object code is generated. Parameters are global and can therefore be referenced throughout the module.

```
PARAMETERS

INTEGER max_number_of_graphs;
  $(The maximum allowable number of "graphs")
```

"Max_number_of_graphs" can be used throughout the module to ensure that the total number of "graphs" never exceeds that value.

The DEFINITIONS paragraph is purely for the convenience of the designer and reader. It is a macro facility. The named object, when it is used in the specification, is replaced by the particular expression which it represents. For example,

```
DEFINITIONS

REAL square_root (REAL x; REAL toller)
  IS $(Returns the square root of a number, x, within a
       given tollerance toller.)
     IF x<0  OR  toller<=0
        THEN ?
        ELSE SOME REAL r |
             r*r-toller<=x AND
             r*r+toller>=x;
```

The name of this particular definition is "square_root." The value defined by it has type REAL. The list following the name contains the formal arguments. Each time "square_root" is used it must be

immediately followed by a list of actual arguments that agree in number and type. Using "square_root" causes the expression following the word IS to be textually substituted for "square_root." "Square_root" can be referred throughout the entire module.

It is apparent that the use of the DEFINITIONS paragraph can extend beyond convenience. It can cut down on errors when definitions are long or used repeatedly. In addition, should the definition of "square_root" be changed, it is changed in one place rather than in each place it is used.

The SOME construct used above will be discussed later. For anyone familiar with a high level programming language, the "square_root" definition is readable with practically no further explanation. The only exception is the symbol "?", which represents the unique value "undefined." "Square_root" evaluates to "undefined" when a square root of a negative number is indicated or if the tollerance is less than or equal to zero.

Objects "x" and "toller" do not have to be declared in the DECLARATIONS paragraph. An object may be bound to a type when it is used. For "x" and "toller", the scope is the definition of "square_root." It is not important while reading a specification to know the scope of each object, just its type. In the example of "square_root", the binding of "x" to type REAL occurs in the definition. If there was no binding of "x" in

"square_root", the reader would have to look elsewhere (perhaps in the DECLARATIONS paragraph) in order to discover the type of "x".

SPECIAL describes abstract machines. In other words it presents the functional capabilities of a computation as opposed to the implementation details. The last and most important paragraph contains a description of the functions available at the interface of the module. It is entitled FUNCTIONS.

The state of a module or abstract machine at any given instant is the value of all objects defined by the module's specification. This state changes when at least one of the objects it manipulates changes. SPECIAL's three types of functions reflect this view of a process. They are VFUN, OFUN and OVFUN. VFUNs (value functions) return a value, but do not change the state of the module. OFUNs (operation functions) cause state changes, but return no value. The last category, OVFUN, causes a state change and returns a value.

Picture a module at some level in a hierarchically designed system. The functions in this paragraph describe the interface between the module and other modules that may reference it. The only communication to this module is accomplished via the functions.

For each function, one may specify conditions under which it is invalid to call the function. These are called EXCEPTIONS and are analogous to exception conditions or instruction traps on computers; for instance,

the division instruction(s) on most computers generate an exception condition on division by zero. For the OFUN functions, which are operations changing the state of a module, the new state must be defined precisely in terms of the old state and the arguments to the new function; the changes in the various components of the state of the machine are given as EFFECTS. For the VFUN functions, which represent components of the state of the machine, the value must be defined precisely. Some VFUN functions represent components of the state not accessible to the user, but necessary to define the semantics of other functions; these are termed HIDDEN. (The example of a stack indicates the need of HIDDEN functions. Though only the top element is accessible to the user, the complete state of the module depends on the whole sequence of items stored in the stack.)

The following example describes the characteristics of the different function types. Although a bit contrived, it will illustrate the main ideas. The example defines a single "register." It can be initialized to an INTEGER value, have another INTEGER value added to it or have twice its value returned. Assume that in the PARAMETERS paragraph, "max_register_value" has been declared as an INTEGER. It is intended to be the maximum absolute value that can be stored in the "register."

```
FUNCTIONS

VFUN register( ) -> INTEGER value;
$(Returns the value stored in the "register".)
```

```
  HIDDEN;
  INITIALLY
    value=?;

OFUN load_register (INTEGER i);
$(Stores I in the "register.")
  EXCEPTIONS
    max_register_value<i;
    max_register_value<-i;
  EFFECTS
    'register( )=i;

OVFUN add (INTEGER i) -> INTEGER value;
$(Adds "i" to the contents of the "register" and returns
  the new value as the value of the function.)
    EXCEPTIONS
      register( )=?;
      max_register_value<register( )+i;
      max_register_value<-(register( )+i);
    EFFECTS
      value='register( );
      'register( )=register( )+i;

VFUN double ( ) -> INTEGER value;
$(Returns double the contents of the "register.")
  EXCEPTIONS
    register( )=?;
  DERIVATION
    register( )*2;
```

The first function is a VFUN called "register," which has no formal arguments. The value returned by this function is an INTEGER as indicated by the information following the symbol "->". "Value" has been bound to type INTEGER. The scope of its binding is the function definition. Corresponding to the discussion of the object "x" in the example of "square-root", "value" could have been bound to INTEGER elsewhere.

After the comment is the reserved word HIDDEN. VFUNs may be hidden as the reserved word indicates or visible. Visible VFUNs are available at the interface whereas hidden VFUNs are not available outside of the module. Hidden VFUNs are used by the designer for

purposes of description.

The reserved word INITIALLY indicates that this VFUN is primitive. This means that an initial value for the VFUN exists. The initial value for "register" is "?" or "undefined." All of the primitive VFUNs together define the state of the module.

The next function, "load_register", is an OFUN; it therefore causes a state change. It takes a single INTEGER parameter. As the comment indicates, the parameter's value is to be placed into the "register."

The next keyword in "load_register" is EXCEPTIONS. An EXCEPTIONS section appears in all except HIDDEN functions. They describe conditions under which the function cannot be called. If any of the statements (exception conditions) in the EXCEPTIONS section are true, then no state change takes place and/or no value is returned. As was previously stated, "max_register_value" is a parameter which represents the maximum absolute value storable in "register." The two exception conditions prevent an integer which is outside the acceptable range from being placed into the "register."

If no exception conditions prevent the function from executing, the EFFECTS section describes the state change. In defining a state change, we must refer to the new values for components (VFUNs) of the state in terms of old values. Placing the symbol quote before the function name "register" is the value in the new

state, i.e. the value returned by a call to "register" after the call to "load_register." Therefore, the effect of this OFUN is that a subsequent call to "register" will return the value "i". Note, the new state of the process is completely represented by the values returned by all primitive VFUNs.

The next function is an example of an OVFUN function. "Accumulate" takes one argument, an INTEGER, and returns an INTEGER value. The comments indicate the effects this function has on the state as well as the value returned. Since this function is visible, there is an EXCEPTIONS section. "Accumulate" cannot effect the state of the process if the contents of "register" is "undefined." This is controlled by the first exception condition. The second and third exception conditions restrict the new value in "register" to be within the acceptable range. Exception conditions have an ordering: the first is checked, then the second and so on. This is unlike the EFFECTS section where all changes occur simultaneously. As a result of a call to "accumulate," the new value returned by "register" is equal to the value returned by "register" before this function call plus "i." Lastly, the value returned is the value that will be returned by the function "register" after the call to "accumulate."

"Double_register" is the last function. It is a VFUN which has an empty argument list and returns an INTEGER value. The only exception condition is that the

"register" cannot be "undefined." Recall that the first VFUN "register" was primitive. Unlike "register," the function "double_register" is derived (as indicated by the keyword DERIVATION). Its value is defined or derived in terms of other VFUNs. The value returned by "double_register" is the current value of "register" times two.

In conclusion, the three types of functions are VFUNs (value functions), OFUNs (operation functions) and OVFUNs (operation and value functions). Each VFUN is either hidden or visible, depending on whether it is available outside of the module. The value returned from a VFUN is derived or primitive. OFUNs and OVFUNs, like visible VFUNs, have EXCEPTIONS or conditions under which they will not be executed. Should no exception condition arise, the effects of an OFUN or OVFUN are described in the EFFECTS section.

A few more language details must be explained. The majority of SPECIAL's constructs are intuitively equivalent to high level language expressions (i.e. IF-THEN-ELSE). The remaining expressions that are particular to the specification language and that appear in the enclosed examples will now be described.

As previously mentioned, type-constructors are used to build new types from primitive ones. Three type-constructors are: VECTOR_OF, SET_OF and STRUCT_OF. Vectors are defined explicitly,

```
VECTOR(2   4   6)
```

or implicitly,

```
VECTOR(FOR j FROM 1 TO 3:  2*j).
```

These two expressions define the same sequence of three numbers. Note that the object "j" in the implicit vector need not be declared as INTEGER; this is assumed. If A is a vector, then LENGTH(A) returns the number of elements in A. The third element in A may be referenced by A[3].

Sets may be defined implicitly such as the following expression:

```
{INTEGER i | i>3 AND i<=7}.
```

This set contains elements 4, 5, 6 and 7. The symbol "|" is read as "such that." The entire expression is read "the set of INTEGER i such that i is greater than 3 and i is less than or equal to 7." CARDINALITY returns the number of elements in the set which is its operand.

The SOME expression returns any value having a given property (or equivalently, satisfying a given predicate. For example,

```
SOME x | T(x)
```

translates as "some x such that T(x) is true."

Assume a new type called "record" is defined by the statement:

```
record: STRUCT_OF(INTEGER identification; INTEGER
value)
```

If the object A is of type "record," then the expression A.value references the second item in the structure. A=STRUCT(1;3) assigns "A" the value of a structure where A.identification equals 1 and A.value equals 3.

A DESIGNATOR is used to define a set of names to reference elements of a new class of objects. The function NEW applies to designators only. NEW(t) generates a name that has never been used before. "t" must have been defined as a designator in the TYPES paragraph by the expression t: DESIGNATOR.

The LET expression provides a local definition of a value. The scope of the definition is the expression

```
LET INTEGER j =
      CARDINALITY({INTEGER k |
            k>0 AND
            k<50})
   IN
        <expression>
```

The object "j" is defined and bound for the duration of the expression following the keyword IN. (Of course, the value of j in the example happens to be 49.)

Lastly, SPECIAL has an expression for universal quantification.

```
FORALL x | T(x): U(x)
```

is read "forall x such that T(x) is true, Q(x) is true."

Appendix II
Stacks

A. Keywords: stack, pushdown, LIFO, LIFO queue

B. References

The concept of a stack is defined in many texts. One is chapter three of Fundamentals of Data Structures, by Ellis Horowitz and Sartaj Sahni, Computer Science Press, Inc., 1976. For ideas on implementing the module, chapters three and four of the same text present several strategies.

C. Hidden Information

This module hides all decisions about implementing a stack. For instance, users of the module cannot answer the following questions: Are the stacks implemented using an array or linked list? Do all stacks share one array or does each have its own array? How many words, bytes, or bits are used per data element in the stack?

D. Description (an aid to understanding the definition, though the definition is the arbitrator of all issues or questions raised)

This module manages any number of stacks up to the implementation constant "maxstacks." The data structure represented by a stack maintains a sequence of items. One can add an item to the sequence using "push", which adds the item at one fixed end of the sequence. An item may be removed from the sequence at that same fixed end using "pop", which besides removing the item, gives its value as the value of the procedure call. One may obtain that value without removing the data item from the sequence via "top." The only other operation is to ask whether there are any elements in the sequence or not, via "empty." The maximum length of any sequence is "maxsize." For this specification, the data items are integers whose absolute value is bounded by "maxelement." New stacks may be created and old stacks released via "create_stack" and "delete_stack."

Note particularly that the function "stack(s)" is HIDDEN. Therefore, it can never be called, nor does it imply an implementation using arrays or sequential memory. It merely indicates the effects of "push" and "pop" on the sequence of items. The specification implies that only the element most recently entered into the data structure may be accessed or

removed; this property has led to the phrase "last-in-first-out" or LIFO.

E. Modifications

Stacks of course do not have to be sequences of integers. The declarations which must be changed for REALs, CHARs, or whatever is desired are marked by comments in the specfication.

For certain applications, one may wish to access any element in the sequence, but add or delete items at only one fixed end. For this, one may add another function:

```
VFUN value(s;INTEGER j)->i;
  $(random access read of data
    items in stack)
  EXCEPTIONS
    stack(s)=?;
    j<1 OR j>size(s);
  DERIVATION
    stack(s)[j];
  $(Note that value(s;1) is the
    first element added to the
    sequence, not the last one.)
```

Another function one may wish to add to the interface is the number of elements in a stack. "Size" in the DEFINITIONS section suggests how to add that feature.

F. Alternatives

In some applications such as implementations of programming languages, a pointer into the stack may be desirable. The pointer may be used as a local reference point for selectively reading values stored in the stack or removing many items simultaneously. Refer to the specification stack1 for such a feature.

```
MODULE stacks


    TYPES

stack_name: DESIGNATOR;


    DECLARATIONS

INTEGER i;  $( This is for a stack of integers. The declarations
             must be changed for a different type of data element.)
stack_name s;


    PARAMETERS

INTEGER max_stack_size $( maximum size of any stack) ,
       max_number_of_stacks  $( maximum number of stacks permitted),
       max_element_value  $( maximum absolute value storable in
                             any stack)
                          $( For a stack of a different data type
                           than INTEGER, this definition must be
                           changed.);


    DEFINITIONS

INTEGER number_of_stacks
   IS CARDINALITY{{ stack_name s | stack(s) ~= ? }};
INTEGER size_of_stack(s) IS LENGTH(stack(s));


    FUNCTIONS


VFUN stack(s) -> VECTOR_OF INTEGER v;
          $( Represents contents of stack s. This declaration of
           INTEGER must be changed for a different type of data
           element.)
    HIDDEN;
    INITIALLY
       v = ?;

VFUN empty(s) -> BOOLEAN b;  $( This function returns true, if
                              stack s is empty, otherwise false.)
    EXCEPTIONS
       stack(s) = ?;
    DERIVATION
       size_of_stack(s) = 0;
```

```
VFUN top(s) -> i;  $( Returns the value most recently pushed onto
                      stack s.)
    EXCEPTIONS
       stack(s) = ?;
       empty(s);
    DERIVATION
       stack(s)[size_of_stack(s)];
          $( One might add another derived VFUN which is just the
           macro size(s) so that the number of elements in
           the stack is accessable.)

OVFUN create_stack() -> s; $( Initializes a new stack.)
    EXCEPTIONS
       number_of_stacks >= max_number_of_stacks;
    EFFECTS
       s = NEW(stack_name);
       'stack(s) = VECTOR();

OFUN delete_stack(s);  $( Delete makes stack s and its contents
                          unavailable.)
    EXCEPTIONS
       stack(s) = ?;
    EFFECTS
       'stack(s) = ?;

OFUN push(s; i);
          $( Push adds i to the stack s, thereby guaranteeing that
           s is not empty, and making i the element returned by top
           (s).)
    EXCEPTIONS
       stack(s) = ?;
       size_of_stack(s) >= max_stack_size;
       i <(0 - max_element_value) OR i > max_element_value;
    EFFECTS
        'stack(s)
       = VECTOR(FOR j FROM 1 TO size_of_stack(s) + 1
                : IF j <= size_of_stack(s)
                   THEN stack(s)[j]
                   ELSE i);

OVFUN pop(s) -> i; $( Removes item last pushed onto stack s.)
    EXCEPTIONS
       stack(s) = ?;
       empty(s);
    EFFECTS
       i = top(s);
        'stack(s)
       = VECTOR(FOR j FROM 1 TO size_of_stack(s) - 1
                : stack(s)[j]);

END_MODULE
```

Appendix III
Queues

A. Keywords: queue, FIFO

B. References

The concept of a queue is defined in many texts. One is chapter three of Fundamentals of Data Structures, by Ellis Horowitz and Sartaj Sahni, Computer Science Press, Inc., 1976. Chapters three and four of the same text present several strategies.

C. Hidden Information

This module hides all decisions about how the queue is represented in memory. For instance, one does not know whether an array is used, a linked list, or some other structure in memory. Perhaps each queue has its own memory space, or perhaps they share a common memory pool. Whether more than one queue element is packed per word is also hidden. Even though the specification appears to imply that the newest element of the queue enters at the left end, this is not necessarily true.

D. Description (an aid to understanding the definition, though the definition is the arbitrator of all issues or questions raised)

The module manages any number of queues up to "max_number_of_queues." A queue is a sequence of data items very much like a line of individuals at a bank teller's window. Items enter the sequence only at the back end of a queue; "enqueue" is the operation to add a data value to the queue. The first value placed in the queue is the first to be removed; the last entered is the last to be removed. (This has led to its name as "first-in-first-out" or FIFO.) "Dequeue" removes one item from the queue, similar to the person at the front of a line being the first to leave. "Dequeue" returns as a value that data item which is removed. To merely read the value which would be removed by the next use of "dequeue", the function "front" is provided; the value remains in the queue, changing nothing.

Only one item may be examined in the queue at any time. Also, the relative order of the elements with respect to one another cannot be changed and is determined solely by the order in which they were enqueued.

To find out whether the queue has any elements in it, "empty" is used; it does not change the queue at all. If you want a new queue, "create_queue" provides as a value the name of a new queue. To release a queue named q, "abolish_queue" is used.

The maximum sequence length is "max_queue_size." In this specification, all values in the queue are integers whose absolute value is bounded by "max_element_value."

Especially note that function "queue(q)" is HIDDEN. Thus, one can never refer to such a function; it is not part of the interface. Though, the specification may seem to suggest an implementation using an array, following that thought would lead to a very inefficient implementation, causing a shift of every queue element with each "enqueue." In fact, all that is implied by the specfication is that the relative order of the queue elements is exactly the order in which elements were enqueued and that only the item available through "front" or "dequeue" may be read.

E. Modifications

If data values other than integers are desired, several declarations, which are commented in the specification, must be changed. If a count of the number of elements in a queue is desired, "size" should be made a VFUN rather than its present status as a DEFINITION.

F. Alternatives

For certain applications, we may wish to assign some importance, weight, or priority to the elements, and order them based on that first, then on order of entry. See the specification of priority_queues for such a data structure.

```
MODULE queues


    TYPES

queue_name: DESIGNATOR;


    DECLARATIONS

INTEGER i;  $( This is for a queue of integers. This declaration
             must be changed for a different type of data element.)
queue_name q;


    PARAMETERS

INTEGER max_queue_size $( maximum size of any queue) ,
       max_number_of_queues $( maximum number of queues) ,
       max_element_value
          $( The maximum absolute value storable in any queue. For
           a queue of a different type other than INTEGER, this
           definition must be changed.);


    DEFINITIONS

INTEGER number_of_queues
   IS CARDINALITY({ queue_name q | queue(q) ~= ? });
INTEGER size(q) IS LENGTH(queue(q));


    FUNCTIONS


VFUN queue(q) -> VECTOR_OF INTEGER s;
          $( This declaration of INTEGER must be changed for a
           different type of data element. This is the sequence
           of elements added to q. The most recently added
           is at queue(q)[1]. The oldest entry is the last
           element in the sequence.)
    HIDDEN;
    INITIALLY
       s = ?;

VFUN empty(q) -> BOOLEAN b;  $( Value is true if queue q is empty,
                               otherwise false.)
    EXCEPTIONS
       queue(q) = ?;
    DERIVATION
       size(q) = 0;
```

```
VFUN front(q) -> i;  $( Returns value least recently enqueued or added
                       onto queue q, and which has also not been
                       subsequently dequeued.)
    EXCEPTIONS
       queue(q) = ?;
       empty(q);
    DERIVATION
       queue(q)[size(q)];
          $( One might add another derived VFUN which is just the
           macro size(q) so that the number of elements in
           any queue can be accessed outside of the module.)

OVFUN create_queue() -> q;  $( Initiates a new queue, returning the
                               name of that queue as the function's
                               value.)
    EXCEPTIONS
       number_of_queues >= max_number_of_queues;
    EFFECTS
       q = NEW(queue_name);
       'queue(q) = VECTOR();

OFUN abolish_queue(q);  $( Makes queue q and any contents
                          unaccessable.)
    EXCEPTIONS
       queue(q) = ?;
    EFFECTS
       'queue(q) = ?;

OFUN enqueue(q; i);
          $( Enqueue adds i to the queue q, thereby guaranteeing
           that q is not empty. i would be returned by front only
           after preceding enqueued values have been removed
           using dequeue.)
    EXCEPTIONS
       queue(q) = ?;
       size(q) >= max_queue_size;
       i <(0 - max_element_value) OR i > max_element_value;
    EFFECTS
        'queue(q)
       = VECTOR(FOR j FROM 1 TO size(q) + 1
                : IF j > 1 THEN queue(q)[j - 1] ELSE i);

OVFUN dequeue(q) -> i;  $( Removes item least recently added by
                          enqueue onto queue q.)
    EXCEPTIONS
       queue(q) = ?;
       empty(q);
    EFFECTS
       i = front(q);
        'queue(q)
       = VECTOR(FOR j FROM 1 TO size(q) - 1: queue(q)[j]);

END_MODULE
```

Appendix IV
Trees

A. Keywords: tree, binary tree, hierarchy, hierarchical data structure

B. References

The concept of a tree is defined in many texts. One is chapter five of Fundamentals of Data Structures, by Ellis Horowitz and Sartaj Sahni, Computer Science Press, Inc., 1976.

C. Hidden Information

This module hides all decisions about how the tree is represented in memory. For instance, one does not know whether an array is used, a linked list, or some other structure. Perhaps each tree has its own memory space, or perhaps they share a common memory pool. Whether more than one tree element is packed per word as well as the specific algorithms for performing inorder, preorder and postorder traversals are also hidden.

D. Description (an aid to understanding the definition, though the definition is the arbitrator of all issues or questions raised)

This module manages any number of binary trees up to "max_number_of_trees." A binary tree is a finite set of zero or more nodes. A non-empty binary tree has a special node known as the root. Up to two nodes can be directly associated with a node. They are called the leftson and the rightson of that node. This node is then designated as the father and no other node may be the father of either the leftson or rightson. Nodes can have a unit of information associated with them. In this module the information is an integer that has a maximum absolute value of "max_value." Finally, the maximum number of nodes on any binary tree is "max_number_of_nodes."

A tree or its contents cannot be accessed directly at the interface. Two HIDDEN functions are provided in order to describe the contents of a tree and to reflect the changes which occur to it. They are "tree" and "node."

The function "create_tree" initiates a new tree. At this point the new tree contains no nodes. "Initialize_root" adds a root node to an empty tree. A leftson or rightson may be associated with an existing node via the functions

"initialize_left_son" and "initialize_right_son."

Each node of a particular tree has an identification associated with it. The functions "father_of_node", "left_son", and "right_son" return the identification for the node having that relationship to the given one. "Set_value" and "value_of_node" store and retrieve node values in the tree. "Exists_left_son" and "exists_right_son" are predicates which return true if a node has a leftson or rightson, respectively. To delete a node the function "delete_node" should be used. "Delete_tree" will delete an entire tree thereby making it and its nodes unavailable. Lastly, three functions are provided for tree traversals. The tree traversals visit each node in the tree exactly once, returning the node identifiers for each node in the order visited. The three orders are preorder, inorder and postorder which correspond to prefix, infix and postfix forms of an expression. The functions are "preorder_traversal", "inorder_traversal" and "postorder_traversal."

E. Modifications

Trees, of course, do not have to contain integers as values of the nodes. The appropriate declarations would have to be changed.

F. Alternatives

In some applications, such as certain sorts, the more general n_ary tree, as opposed to the binary tree, is desirable.

```
MODULE trees

 $( It is probably best to study the specification in the
  following order:  node_structure under TYPES first;
  create_tree and initialize_root for creating a new tree;
  initialize_left_son or initialize_right_son to set up
  children;  father_of_node, left_son, and right_son to
  find a node having that relationship to the given one;
  set_value and value_of_node to store and retrieve values
  in the tree; and the remaining features.)


    TYPES

tree_name: DESIGNATOR;
node_structure:
STRUCT_OF(INTEGER father;
          INTEGER value;
          INTEGER leftson;
          INTEGER rightson);
          $( For a tree holding a different data type other than
           INTEGER, the declaration for value must be changed.)
          $( A node has four pieces of information potentially:
           the identity of the father, the value stored at the node,
           the identity of a leftson, and the identity of a rightson.
           This fact is indicated by our use of the STRUCTure
           construct; the specification refers to any of the four
           pieces of information in a structure s by writing s.father
           for the father information, s.value for the value, etc.)


    DECLARATIONS

INTEGER id, i, j;
tree_name t;
node_structure n;


    PARAMETERS

INTEGER max_number_of_trees, $( maximum allowable number of trees)
       max_number_of_nodes,  $( maximum allowable number of nodes
                                on a tree)
      max_value
          $( The maximum absolute value storable in any node. For a
           data type other than INTEGER, this declaration must be
           changed.);
```

```
    DEFINITIONS

SET_OF INTEGER nodes_in_tree(t)
   IS { INTEGER id | node(t, id) ~= ? };
INTEGER size_of_tree(t) IS CARDINALITY(nodes_in_tree(t));
INTEGER number_of_trees
   IS CARDINALITY({ tree_name t | tree(t) ~= ? });
VECTOR_OF INTEGER combine_vectors(VECTOR_OF INTEGER v1;
                                  VECTOR_OF INTEGER v2;
                                  VECTOR_OF INTEGER v3)
   IS  $( A vector which is the concatenation of vectors
       v1, v2 and v3.)
     VECTOR(FOR j
              FROM 1
              TO LENGTH(v1) + LENGTH(v2) + LENGTH(v3)
                : IF j <= LENGTH(v1)
                   THEN v1[j]
                   ELSE IF j <= LENGTH(v1) + LENGTH(v2)
                         THEN v2[j - LENGTH(v1)]
                         ELSE v3[j -(LENGTH(v1) + LENGTH(v2))]);
VECTOR_OF INTEGER preorder(t; id)
   IS  $( A vector of integers, which represents all node
       identifiers for the tree t, in a preorder traversal. id is
       the identifier of the root node of the tree.)
     IF id = ?
       THEN VECTOR()
       ELSE combine_vectors(VECTOR(id),
                            preorder(t, node(t, id).leftson),
                            preorder(t, node(t, id).rightson));
VECTOR_OF INTEGER inorder(t; id)
   IS  $( A vector of integers, which represents all node
       identifiers for the tree t, in an inorder traversal. id is
       the identifier of the root node of the tree.)
     IF id = ?
       THEN VECTOR()
       ELSE combine_vectors(inorder(t, node(t, id).leftson),
                            VECTOR(id),
                            inorder(t, node(t, id).rightson));
VECTOR_OF INTEGER postorder(t; id)
   IS  $( A vector of integers, which represents all node
       identifiers for the tree t, in a postorder traversal. id is
       the identifier of the root node of the tree.)
     IF id = ?
       THEN VECTOR()
       ELSE combine_vectors(postorder(t, node(t, id).leftson),
                            postorder(t, node(t, id).rightson),
                            VECTOR(id));
INTEGER root_id(t)
   IS $( The identifier of the root node for tree t.)
      SOME INTEGER i | father_of_node(t, i) = 0;
```

```
    FUNCTIONS


VFUN node(t; i) -> n;  $( Returns the node n in tree t with
                          identifier i.)
    HIDDEN;
    INITIALLY
       node(t, i) = ?;

VFUN tree(t) -> BOOLEAN b;  $( Predicate, which returns true if
                               tree t exists, otherwise false.)
    HIDDEN;
    INITIALLY
       FALSE;

VFUN father_of_node(t; i) -> id;
          $( Returns the identifier for the node which is the
           father of node i on tree t. Returns zero for the root.)
    EXCEPTIONS
       NOT tree(t);
       node(t, i) = ?;
    DERIVATION
       node(t, i).father;

VFUN left_son(t; i) -> id;  $( Returns the identifier for the node
                               which is the leftson of node i on
                               tree t.)
    EXCEPTIONS
       NOT tree(t);
       node(t, i) = ?;
       NOT exists_left_son(t, i);
    DERIVATION
       node(t, i).leftson;

VFUN right_son(t; i) -> id;  $( Returns the identifier for the node
                                which is the rightson of node i on
                                tree t.)
    EXCEPTIONS
       NOT tree(t);
       node(t, i) = ?;
       NOT exists_right_son(t, i);
    DERIVATION
       node(t, i).rightson;

VFUN exists_left_son(t; i) -> BOOLEAN b;
          $( Predicate which returns true, if node i on tree t has
           a leftson, otherwise false.)
    EXCEPTIONS
       NOT tree(t);
       node(t, i) = ?;
    DERIVATION
       node(t, i).leftson ~= ?;
```

```
VFUN exists_right_son(t; i) -> BOOLEAN b;
          $( Predicate which returns true, if node i on tree t has
           a rightson, otherwise false.)
    EXCEPTIONS
       NOT tree(t);
       node(t, i) = ?;
    DERIVATION
       node(t, i).rightson ~= ?;

VFUN value_of_node(t; i) -> j;
          $( Returns the stored value of node i on tree t. The type
           of j must be changed for a data type other than INTEGER.)
    EXCEPTIONS
       NOT tree(t);
       node(t, i) = ?;
       node(t, i).value = ?;
    DERIVATION
       node(t, i).value;

OFUN delete_node(t; i); $( Deletes leaf node i on tree t.)
    EXCEPTIONS
       NOT tree(t);
       node(t, i) = ?;
       exists_left_son(t, i);
       exists_right_son(t, i);
    EFFECTS
       IF father_of_node(t,i) = 0
         THEN $( The node is the root node.) 'node(t, i) = ?
         ELSE IF node(t, father_of_node(t,i)).leftson = i
              THEN $( The node is the leftson of some node.)
                  'node(t, father_of_node(t, i)).leftson = ?
                 AND 'node(t, i) = ?
              ELSE $( The node is the rightson of some node.)
                  'node(t, father_of_node(t, i)).rightson = ?
                 AND 'node(t, i) = ?;

OFUN delete_tree(t); $( Deletes an entire tree.)
    EXCEPTIONS
       NOT tree(t);
    EFFECTS
       'tree(t) = FALSE;
       FORALL i | node(t, i) ~= ?: 'node(t, i) = ?;

OVFUN create_tree() -> t;  $( Initiates a new tree, returning the
                            name of that tree as the function's
                            value.)
    EXCEPTIONS
       max_number_of_trees <= number_of_trees;
    EFFECTS
       t = NEW(tree_name);
       'tree(t) = TRUE; $( t is now a valid tree_name, but the
                          tree is empty for now)
```

```
OVFUN initialize_root(t) -> id;
          $( Allows a value, leftson, or rightson to be
           associated with the root of tree t. The value returned
           is the identifier for the root. The father of the root
           is set to zero.)
    EXCEPTIONS
       NOT tree(t);
       size_of_tree(t) ~= 0;
    EFFECTS
       LET INTEGER j | j ~= 0 AND node(t, j) = ?
         $( j was not in use as a node name)
        IN 'node(t, j) = STRUCT(0, ?, ?, ?) AND id = j;
          $( j becomes a root since its father is 0. It has no
            leftson, no rightson, nor any value stored at it.)

OVFUN initialize_left_son(t; i) -> id;
          $( Adds a new node to tree t, returning the identifier
           for that node as the function's value. The new node is the
           leftson of node i.)
    EXCEPTIONS
       NOT tree(t);
       node(t, i) = ?;
       exists_left_son(t, i);
       size_of_tree(t) >= max_number_of_nodes;
    EFFECTS
       LET INTEGER j | j ~= 0 AND node(t, j) = ?
         $( j was not in use as a node name)
        IN  'node(t, j) = STRUCT(i, ?, ?, ?)   $( j has a father i;
                                                  its leftson, rightson,
                                                  and stored value are all
                                                  undefined)

          AND 'node(t, i).leftson = j   $( this defines that i has j
                                           as its leftson)

          AND id = j;
```

```
OVFUN initialize_right_son(t; i) -> id;
          $( Adds a new node to tree t, returning the identifier
           for that node as the function's value. The new node is the
           rightson of node i.)
    EXCEPTIONS
       NOT tree(t);
       node(t, i) = ?;
       exists_right_son(t, i);
       size_of_tree(t) >= max_number_of_nodes;
    EFFECTS
       LET INTEGER j | j ~= 0 AND node(t, j) = ?
         $( j was not in use as a node name)
        IN  'node(t, j) = STRUCT(i, ?, ?, ?)   $( j has a father i;
                                                  its leftson, rightson,
                                                  and stored value are all
                                                  undefined)

            AND 'node(t, i).rightson = j   $( this defines that i has j
                                              as its rightson)

           AND id = j;

OFUN set_value(t; i; j);  $( This must be changed for a data type
                           other than INTEGER for value. Sets the
                           value of node i on tree t to j.)
    EXCEPTIONS
       NOT tree(t);
       node(t, i) = ?;
       j < 0 - max_value OR j > max_value;
    EFFECTS
       'node(t, i).value = j;

VFUN preorder_traversal(t) -> VECTOR_OF INTEGER v;
          $( Returns a vector of integers, which represents all
           node identifiers for the tree t, in a preorder
           traversal.)
    EXCEPTIONS
       NOT tree(t);
       size_of_tree(t) = 0;
    DERIVATION
       preorder(t, root_id(t));

VFUN inorder_traversal(t) -> VECTOR_OF INTEGER v;
          $( Returns a vector of integers, which represents all
           node identifiers for the tree t, in an inorder
           traversal.)
    EXCEPTIONS
       NOT tree(t);
       size_of_tree(t) = 0;
    DERIVATION
       inorder(t, root_id(t));
```

```
VFUN postorder_traversal(t) -> VECTOR_OF INTEGER v;
          $( Returns a vector of integers, which represents all
           node identifiers for the tree t, in a postorder
           traversal.)
    EXCEPTIONS
       NOT tree(t);
       size_of_tree(t) = 0;
    DERIVATION
       postorder(t, root_id(t));

END_MODULE
```

Appendix V
Character Oriented Editor

A. Key Words: editor, text editor, character editor, file

B. References

This module was modeled after TECO, an editor found on the DEC-10 as well as other systems. A survey of on-line editors along with examples can be found in the article On-line Text Editing: A Survey, by Andries Van Dam and David E. Rice, Computing Surveys, Vol. 3, No. 3, September, 1971.

C. Hidden Information

The module hides all decisions about how the editor is implemented. For instance, users of the module cannot assertain how the edit file is stored. Is it in an array, a linked list or some other data structure?

D. Description (an aid to understanding the definition, though the definition is the arbitrator of all issues or questions raised)

This module represents the functional capabilities of an editor. It is assumed that an intermediary level for processing a command systax exists between this and the user. The edit file is made up of a sequence of characters. At no time can the total number of characters in the file exceed "max_file_size." A line is a sequence of characters bracketed by the implementation parameter "line_end." The first and last lines of a file are bracketed by only one "line_end" and either the beginning or the end of the file. There are no explicit line numbers. Operations on the file are relative to a current position.

The file cannot be accessed directly. It is only through the HIDDEN function "file" that the text may be retrived. This function is neccessary in order to describe changes which effect the file. The current position is represented by the HIDDEN function "pointer."

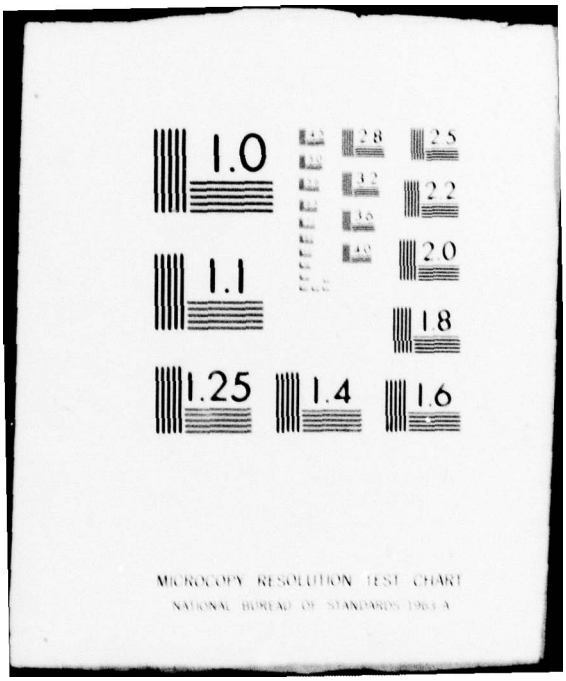
At the onset, the file is closed and uninitialized. The function "open_file" must be called before editing operations may begin. "Open" is a predicate which returns true when the file has been initialized. "Close_file" terminates an editing session.

AD-A068 970 DELAWARE UNIV NEWARK DEPT OF COMPUTER AND INFORMATI--ETC F/G 9/2
TOWARD A LIBRARY OF FORMAL DESIGNS OF SOFTWARE.(U)
1979 R M WEISCHEDEL, L SALSBURG AFOSR-78-3539

UNCLASSIFIED AFOSR-TR-79-0528 NL

2 OF 2
AD
A068970

END
DATE
FILMED
6-79
DDC


1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

The only function that allows a view of the file outside of this module is "display_lines." "Display_lines" makes lines visible relative to the position that the pointer references at the time of the function call. The remaining functions don't return a value, they only have effects on the pointer and the contents of the edit file.

The following functions strictly move the pointer. They are, "move_pointer_i_characters", "move_pointer_over_i_lines", "move_to_beginning_of_file", "move_to_end_of_file" and "scan." "Move_pointer_i_characters" moves the pointer foward or backward over characters. "Move_pointer_over_i_lines" moves the pointer foward or backward over lines. "Move_to_beginning_of_file" positions the pointer to the first character of the file, and "move_to_end_of_file" positions the pointer one past the end of the file. Lastly, "scan" is used to position the pointer to the character following a given number of occurrences of a character string.

The rest of the functions alter the appearance of the file as well as possibly move the pointer. They are, "delete_characters", "delete_lines", "delete_entire_file", "insert_characters", and "replace." "Delete_characters" and "delete_lines" delete characters and lines from the file, respectively. The entire file can be deleted via "delete_entire_file." "Insert_characters" is used to add text to the edit file. Lastly, "replace" substitutes one string of characters for another.

E. Modifications

One way to make this editor more robust would be to add a facility for defining editing macros. In this manner, new editing primitives could be created from existing ones.

F. Alternatives

Rather than having a pointer into the file, lines of text could have a reference number. Locations for changes to the edit file could be specified by using this line number as opposed to positioning the pointer. See the specification of line_oriented_editor.

```
MODULE character_oriented_editor

 $( It is probably best to study the specification in the following
  order: the PARAMETERS line_end and max_file_size;  the VFUNs file
  and pointer;  the DEFINITIONS of file_size, number_of_lines,
  end_of_line, start_of_line, line, and current_line#;  and the
  other FUNCTIONS, reading the remaining DEFINITIONS as they are used
  in FUNCTIONS.)


    TYPES

one_line: VECTOR_OF CHAR;


    DECLARATIONS

INTEGER i;


    PARAMETERS

CHAR line_end;  $( character in the file indicating the end of a
                   line)
INTEGER max_file_size;  $( maximum number of characters permissible
                          in a file)


    DEFINITIONS
          $( A line is a sequence of characters bracketed by
           line_ends; the line_end characters themselves are not
           part of the line. The first and last line of a file are
           bracketed by only one line_end and either the beginning
           or end of the file.)

INTEGER file_size IS LENGTH(file());  $( The total number of
                                        characters in the file,
                                        including line_ends.)

INTEGER number_of_lines
   IS 1 + CARDINALITY({ INTEGER k | file()[k] = line_end });
          $( The total number of lines in the file is always one
           greater than the number of line_ends, since line_ends
           act as separators between lines.)

INTEGER end_of_line(i)
   IS IF i = number_of_lines
       THEN file_size $( The case of the last line.)
       ELSE SOME INTEGER j | file()[j] = line_end
                          AND CARDINALITY({ INTEGER k |
                           k < j AND file()[k] = line_end }) = i - 1;
          $( The clause about the CARDINALITY says that there are
           exactly i-1 line_ends prior to position j.)
```

```
INTEGER start_of_line(i)
   IS IF i = 1 THEN 1 ELSE 1 + end_of_line(i - 1);
         $( Gives the position in the file of the first character
          in line i)

one_line line(i)
   IS VECTOR(FOR j FROM start_of_line(i) TO end_of_line(i)
            : file()[j]); $( The character string forming line i)

INTEGER current_line#
   IS 1 + CARDINALITY({ INTEGER j | file()[j] = line_end
                                    AND j < pointer() });
         $( This is the line number of the line into which pointer
          is pointing. It is always true that the number of
          line_ends intervening between the beginning of the file
          and any particular character position is one less than
          the line number of the line containing the particular
          character position. Note that by this definition, if the
          pointer is at a line_end, current_line# is that of the
          line terminated by the line_end.)

BOOLEAN enough_characters(i)
   IS  (i > 0 AND i + pointer() <= file_size + 1)
         $( The pointer may point one position past the last
          character in the file.)
     OR(i < 0 AND pointer() + i >= 1)
    OR i = 0;  $( This is true if there are enough characters
                 relative to the pointer position to do the desired
                 character operation.)

BOOLEAN enough_lines(i)
   IS  (i > 0 AND current_line# + i <= number_of_lines + 1)
      OR(i < 1 AND current_line# + i >= 1)
    OR i = 0;  $( This is true if there are enough lines relative
                 to the pointer position to perform the
                 delete_lines operation.)

VECTOR_OF CHAR character_delete(INTEGER start_position;
                                INTEGER number_to_delete;
                                VECTOR_OF CHAR string)
         $( This returns a sequence of characters identical to
          string except that a substring has been deleted. The
          substring contains number_to_delete characters and
          starts with start_position.)
   IS VECTOR(FOR i FROM 1 TO LENGTH(string) - number_to_delete
            : IF i < start_position
              THEN string[i]
              ELSE string[i + number_to_delete]);
```

```
VECTOR_OF CHAR character_insert(INTEGER insertion_point;
                                VECTOR_OF CHAR insertion_string;
                                VECTOR_OF CHAR original_string)
   IS  $( This returns a sequence of characters identical to
        original_string except that string is added just before
        insertion_point.)
     VECTOR(FOR i
            FROM 1
            TO LENGTH(insertion_string) + LENGTH(original_string)
             : IF i < insertion_point
                THEN original_string[i]
                    $( The portion before the added characters.)
                ELSE IF i < insertion_point + LENGTH(insertion_string)
                      THEN insertion_string[i - insertion_point + 1]
                        $( For i=insertion_point, the character
                         in the resulting string must be
                         insertion_string[i].)
                      ELSE original_string[i -
                                           LENGTH(insertion_string)])
                        $( The portion after the added characters.);

INTEGER not_enough_matches IS 0;

INTEGER no_more_matches IS 0;  $( Note, it is the same value as
                                  not_enough_matches.)

BOOLEAN at_end(VECTOR_OF CHAR file_string;
               INTEGER starting_point;
               VECTOR_OF CHAR string)
   IS     $( True, if there are less than the length of string
          characters from the starting_point through to the end of
          the file_string. True indicates that there can not be a
          match of the contents of string in any substring of
          file_string beginning at starting_point.)
      LENGTH(string) > LENGTH(file_string) - starting_point + 1
          $( The expression LENGTH (file_string) -starting_point+1
          is the number of characters from starting_point through
          to the end of file_string.);

BOOLEAN at_match(VECTOR_OF CHAR file_string;
                 INTEGER starting_point;
                 VECTOR_OF CHAR string)
   IS  $( True, if there is a match of string in file_string
        beginning at starting_point.)
      NOT at_end(file_string, starting_point, string)
     AND (FORALL INTEGER k |
                k > 0
               AND k <= LENGTH(string):
                      string[k] = file_string[starting_point + k - 1]);
```

```
INTEGER next_match(VECTOR_OF CHAR file_string;
                   INTEGER starting_point;
                   VECTOR_OF CHAR string)
   IS  $( The position of the next match of string in file_string
        which occurs on or after starting_point. If there is no
        match, then not_enough_matches is returned.)
      IF at_end(file_string, starting_point, string)
        THEN not_enough_matches
        ELSE IF at_match(file_string, starting_point, string)
             THEN starting_point
             ELSE next_match(file_string, starting_point + 1, string);

INTEGER position_after_ith_match(VECTOR_OF CHAR file_string;
                                 INTEGER starting_point;
                                 VECTOR_OF CHAR string_to_match;
                                 INTEGER i)
   IS  $( The position just after the ith match of string_to_match
        in file_string. The i matches must all start on or after
        starting_point.)
      IF i = 0
       THEN starting_point
       ELSE IF next_match(file_string, starting_point, string_to_match)
                                 = not_enough_matches
            THEN not_enough_matches
            ELSE position_after_ith_match(file_string,
                                          next_match(file_string,
                                                     starting_point,
                                                     string_to_match)
                                           + LENGTH(string_to_match)
                                             $( The position following
                                              the next occurrence of a
                                              match of string which
                                              is on or after
                                              starting_point),
                                          string_to_match, i - 1);

VECTOR_OF CHAR result_of_substitutions(VECTOR_OF CHAR original;
                                       VECTOR_OF CHAR old;
                                       VECTOR_OF CHAR new;
                                       INTEGER starting_point)
   IS     $( A new string of characters where each occurrence of
          the contents of old in original is replaced by the
          contents of new. Starting_point references the next
          character in original where a match may begin.)
      IF at_end(original, starting_point, old)
        THEN original
        ELSE IF at_match(original, starting_point, old)
             THEN LET VECTOR_OF CHAR updated_original =
                    character_insert(starting_point, new,
                                     character_delete(starting_point,
                                                      LENGTH(old),
                                                      original))
                    $( The result of one substitution can be viewed as
                     inserting the new after deleting the old.)
                  IN result_of_substitutions(updated_original, old, new,
                                             starting_point + LENGTH(new)
                    $( The position after the newly inserted string.))
             ELSE result_of_substitutions(original, old, new,
                                          starting_point + 1);
```

```
INTEGER position_after_substitutions(VECTOR_OF CHAR original;
                                     VECTOR_OF CHAR old;
                                     VECTOR_OF CHAR new;
                                     INTEGER starting_point;
                                     INTEGER number_of_matches)
   IS     $( If all occurrences of old in original starting on or
           after starting_point were replaced by new, the position
           following the last substitution would equal the value
           returned by this definition. Number_of_matches is used
           to keep track of the number of matches found.)
     IF next_match(original, starting_point, old) = no_more_matches
       THEN starting_point
              + number_of_matches * (LENGTH(new) - LENGTH(old))
          $( Starting_point has either its original value when the
           definition was first used or points to the position after the
           last occurrence of a match of old that is on or after the
           original starting_point. To find the location of the
           position after the last substitution of new for old,
           this value must be shifted. The shift is the number
           of characters added or deleted from the file as the
           result of a substitution times the number of
           substitutions. LENGTH (old) - LENGTH (new) reflects
           the number of characters added or deleted from
           the file as a result of a substitution.)
       ELSE IF at_match(original, starting_point, old)
              THEN position_after_substitutions(original, old, new,
                                                starting_point
                                                 + LENGTH(old)
                                                   $( Position where
                                                    another match
                                                    may start.),
                                                number_of_matches + 1)
             ELSE position_after_substitutions(original, old, new,
                                                starting_point + 1,
                                                number_of_matches);


    FUNCTIONS


VFUN file() -> VECTOR_OF CHAR vc;  $( Represents text to be edited.)
    HIDDEN;
    INITIALLY
       vc = ?;

VFUN open() -> BOOLEAN b;  $( Predicate, which returns true if file
                              is open and therefore ready for
                              editing.)
    HIDDEN;
    INITIALLY
       FALSE;

VFUN pointer() -> INTEGER p;  $( Pointer into file giving the
                                 current position, from which most
                                 editing operations are defined.)
    HIDDEN;
    INITIALLY
       p = 1;
```

```
OFUN open_file();  $( Defines the form of the input file, upon
                      which all editing operations are performed.)
    EXCEPTIONS
       open();
    EFFECTS
       'open() = TRUE;
        'file()
       = VECTOR(FOR j
                   FROM 1
                   TO SOME INTEGER k | k >= 0 AND k <= max_file_size
                 : SOME CHAR c | TRUE);

OFUN close_file();
          $( Terminates an editing session. If further editing is
           to be done, an open_file must be performed. In
           integrating this module into a complete file package,
           the definitions of open_file and close_file must be
           modified to reflect (a) that close_file writes the
           modified file to disk and (b) the definition, perhaps
           via operating system commands, of which file is opened
           and of exceptions corresponding to not finding a file by
           that name.)
    EXCEPTIONS
       NOT open();
    EFFECTS
       'open() = FALSE;

VFUN display_lines(i) -> VECTOR_OF CHAR vc;
          $( This 'prints' lines or makes them visible relative to
           the current position. If i>0, then the current line
           starting at the current position will be displayed, as
           well as i-1 lines following the current one. If i<=0,
           then the current line up to the position before current
           position is displayed as well as the absolute value of i
           lines preceding the current line. Notice that the value
           returned by display_lines is just the sequence of
           characters including line_ends as they appear in the
           file; if the lines are to be truly printed, the line_end
           characters must be used to generate the appropriate
           control characters for the particular output device
           involved.)
    EXCEPTIONS
       NOT open();
       NOT enough_lines(i);
    DERIVATION
       IF i > 0
         THEN VECTOR(FOR j
                       FROM pointer()
                       TO end_of_line(current_line# + i - 1)
                     : file()[j])
         ELSE VECTOR(FOR j
                       FROM start_of_line(current_line# + i)
                       TO pointer() - 1
                     : file()[j])
          $( For some applications, it may be preferable to have
           display_lines return a sequence of lines rather than a
           string of characters with the line boundaries as
           characters. To do this, the value returned should be
           VECTOR_OF one_line; the vector is easily defined using
           the definitions of line (i) and current_line#.);
```

```
OFUN move_pointer_i_characters(i);
          $( If i>0, then advance the pointer i characters forward.
           If i<0, then move the pointer back over the absolute
           value of i characters. If i=0, then no change occurs. In
           moving the pointer foward, one is allowed to move it one
           position after the last character of the file.)
    EXCEPTIONS
       NOT open();
       NOT enough_characters(i);
    EFFECTS
       'pointer() = pointer() + i;

OFUN move_pointer_over_i_lines(i);
          $( This positions the pointer to the beginning of a line
           i lines from the current pointer. That is, if i=0, the
           pointer is moved to the beginning of the current line.
           If i>0, the pointer is moved to the beginning of the ith
           line after the current one. If i<0, the pointer is moved
           to the beginning of the absolute value of i lines before
           the current line.)
    EXCEPTIONS
       NOT open();
       NOT enough_lines(i);
    EFFECTS
       'pointer() = start_of_line(current_line# + i);

OFUN move_to_beginning_of_file();  $( Moves pointer to the first
                                      character of the file.)
    EXCEPTIONS
       NOT open();
    EFFECTS
       'pointer() = 1;

OFUN move_to_end_of_file();  $( Set the pointer to the position
                               after the last character in the file.)
    EXCEPTIONS
       NOT open();
    EFFECTS
       'pointer() = file_size + 1;

OFUN delete_characters(i);
          $( If i>0, then the character indicated by pointer and
           the i-1 characters following it are deleted;
           additionally, pointer points to the next remaining
           character. If i<0, then the absolute value of i
           characters preceding the pointer are deleted and the
           pointer points to the same character as before. If i=0,
           this has no effect.)
    EXCEPTIONS
       NOT open();
       NOT enough_characters(i);
    EFFECTS
       IF i >= 0
         THEN 'file() = character_delete(pointer(), i, file())
         ELSE  'file()
                 = character_delete(pointer() + i, 0 - i, file())
               AND 'pointer() = pointer() + i;
```

```
OFUN delete_lines(i);
          $( If i>0, then all characters on the current line at or
           after pointer are deleted; additionally, the next i-1
           lines are deleted, and the pointer points to the first
           character after the deleted ones. If i<=0, then all
           characters of the current line preceding the pointer are
           deleted; additionally, the absolute value of i lines
           preceding current line are also deleted. In the case of
           i<=0, the pointer will continue to point to the same
           character as before the delete.)
    EXCEPTIONS
       NOT open();
       NOT enough_lines(i);
    EFFECTS
       IF i > 0
         THEN 'file()
             = character_delete(pointer(),
                                end_of_line( current_line# + i - 1)
                                  - pointer() + 1
                                    $(The number of characters to
                                      be deleted), file())
           $( The value of pointer need not be changed. The pointer
            afterwards either is just past the end of the file if
            all lines to the end of the file were deleted, or is at
            the first character of the next non_deleted line.)
          ELSE  'file()
              = character_delete(start_of_line(current_line# + i),
                                 pointer()
                                  - start_of_line(current_line# + i)
                                   $( The number of characters to be
                                    deleted.),
                                 file())
            AND 'pointer() = start_of_line(current_line# + i);

OFUN delete_entire_file();  $( This deletes the edited copy of the
                               file.)
    EXCEPTIONS
       NOT open();
    EFFECTS
       'file() = VECTOR();
       'pointer() = 1;

OFUN insert_characters(VECTOR_OF CHAR string);
          $( This operation adds the string just before the
           pointer. However, the pointer continues to point to the
           same character. String may in fact include line_ends;
           therefore adding several lines to a file. If one wants
           to add new lines to the end of the file, one should move
           the pointer to the position after the last character in
           the file; in this case, string should begin with a
           line_end if the last character in the file is not a
           line_end.)
    EXCEPTIONS
       NOT open();
       file_size + LENGTH(string) > max_file_size;
    EFFECTS
       'file() = character_insert(pointer(), string, file());
       'pointer() = pointer() + LENGTH(string);
```

```
OFUN scan(VECTOR_OF CHAR string; INTEGER i);
          $( This operation moves the pointer to just after the ith
           occurrence of string occurring at or after the pointer
           in the file.)
    EXCEPTIONS
       NOT open();
       LENGTH(string) = 0;
       i <= 0;
        position_after_ith_match(file(), pointer(), string, i)
       = not_enough_matches;
    EFFECTS
        'pointer()
       = position_after_ith_match(file(), pointer(), string, i);

OFUN replace(VECTOR_OF CHAR old; VECTOR_OF CHAR new);
          $( This substitutes the new string for the old string
           whenever old occurs starting at or after the pointer.
           Additionally, the pointer is positioned to the character
           following the last substitution.)
    EXCEPTIONS
       NOT open();
       LENGTH(old) = 0;
        max_file_size
       < LENGTH(result_of_substitutions(file(), old, new, pointer()));
    EFFECTS
        'file()
       = result_of_substitutions(file(), old, new, pointer());
        'pointer()
       = position_after_substitutions(file(), old, new, pointer(),
                                      0);

END_MODULE
```